Escuela Politécnica Superior

Manipulación de objetos mediante un brazo robótico usando técnicas de aprendizaje por refuerzo

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Jordi Termes Sabater

Tutor/es:
Antonio Jorge Pertusa Ibáñez

Diciembre 2019

Universitat d'Alacant
Universidad de Alicante

Manipulación de objetos mediante un brazo robótico usando técnicas de aprendizaje por refuerzo

Autor

Jordi Termes Sabater

Tutor

Antonio Pertusa Ibáñez

Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Robótica

Escuela Politécnica Superior

ALICANTE, 9 de diciembre de 2019

Preámbulo

Motivación

Este trabajo nace de la pasión, sembrada a lo largo de todo el grado, por profundizar en la investigación de la Inteligencia Artificial.

Asignaturas como Visión por computador, Sistemas inteligentes y Sistemas de percepción me han abierto la puerta a un campo muy amplio e interesante del que todavía queda mucho por explotar. A esto se suma el hecho de que las nuevas tendencias tecnológicas se centran cada vez más en implementar sistemas inteligentes en los dispositivos electrónicos como *smartphones*, vehículos autónomos, *smartwatches* y otros *wearables*, por no hablar de los complejos programas inteligentes que desarrollan las empresas para el análisis de datos, seguridad, etc.

La motivación de este proyecto es ser capaz de unir los conocimientos adquiridos a lo largo de todo el grado y darle forma en la resolución de un problema de carácter real. Pasando por todos los puntos, desde la comprensión de los algoritmos de aprendizaje artificial, su matemática, las herramientas que nos permiten implementarlos y cómo hacerlo, exponerlos ante distintas situaciones y ser capaz de comprender los resultados, extraer conclusiones y averiguar las ventajas que nos ofrecen unos algoritmos frente a otros.

Por supuesto, como estudiante de Ingeniería Robótica, no había mejor forma de aplicar la inteligencia artificial que un brazo robótico articulado. De este modo, realizamos un proyecto que permita a este sistema ser capaz de aprender a moverse en busca de un objetivo concreto, con el incentivo extra de, mediante trabajos futuros, llegar a implementarlo en un robot real.

Objetivos

El objetivo general de este trabajo es que al final del mismo, tanto el autor como el lector hayan comprendido los principios y funcionamiento del aprendizaje por refuerzo (*Reinforcement Learning*), empleando estos en el control de un brazo robótico y obteniendo conclusiones sobre cuáles son los algoritmos que nos proporcionan mejores resultados en menor tiempo de entrenamiento para diversas tareas de dificultad progresiva.

Entrando más en detalle, podemos definir la siguiente lista de objetivos:

- Entender los principios y funcionamiento de *Reinforcement Learning* (RL) y *Deep Reinforcement Learning* (DRL).
- Realizar un estudio de los algoritmos más importantes que conforman el estado del arte de RL y DRL, para poder ser usados posteriormente en la experimentación.
- Aprender a trabajar con un *hardware* capaz de asumir la alta carga computacional que normalmente acarrea el proceso de entrenamiento en sistemas de inteligencia artificial, en tiempos de espera asequibles.
- Instalar el *software* necesario que sirva de base para que el alumno pueda realizar la experimentación sin indagar demasiado en conceptos que escapan de su nivel en la materia. Si bien, el alumno deberá comprender y aprender a utilizar los programas y gran parte del código implementado, además de ser capaz de realizar cambios en la implementación para adaptarlo a las tareas propuestas en este trabajo.
- Conseguir que un brazo robótico aprenda a moverse para realizar diferentes tareas de complejidad progresiva y analizar los resultados aplicando los diferentes algoritmos estudiados.
- Ser capaz de trabajar con la búsqueda de hiperparámetros y conseguir valores adecuados para ellos.
- A partir de los conocimientos adquiridos, ser capaz de analizar, comprender y justificar los resultados de los experimentos, obteniendo conclusiones y propuestas de trabajo futuro.

Agradecimientos

Me gustaría agradecerle de todo corazón a aquellas personas que han estado a mi lado luchando y apoyándome, no solo durante la realización de este trabajo, sino durante toda mi vida.

Este proyecto no es más que una pequeña representación de lo que el esfuerzo y las ganas de aprender pueden llegar a alcanzar. Espero seguir trabajando en el futuro con el mismo esfuerzo y dedicación que lo he hecho en este proyecto. Pero evidentemente, para cumplir con ello será necesario la ayuda de amigos, familiares, profesores y demás referentes tan importantes como los he tenido en esta ocasión.

Muchas gracias a mi tutor Antonio Pertusa por haber estado disponible siempre que me ha sido necesario y haber aprendido conmigo de este trabajo. Ambos sabemos con todas las dificultades que nos hemos encontrado hasta darle un rumbo fijo a este trabajo, y las tantas más que han surgido durante su realización.

Me gustaría agradecerles a mis amigos y compañeros de clase, que unos a otros nos hemos ayudado y hecho piña para que nuestra transición por la universidad fuese más amena. Y que no se nos olvide lo más importante, todo lo que hemos vivido fuera de la universidad. A vosotros os debo muchas de las mejores experiencias que he vivido. Gracias por enseñarme a disfrutar del momento.

Por último pero no menos importante, gracias a mis padres porque ellos me lo han dado todo, a mis hermanas por enseñarme a compartir, y a mis abuelos por su amor incondicional. Este trabajo va dedicado a todos vosotros.

Si la situación te viene grande... CRECE!

100 %

Contenidos

Capítulo 1

Introducción

Este proyecto profundiza en la comprensión y desarrollo de diferentes técnicas de Aprendizaje por Refuerzo y Aprendizaje por Refuerzo Profundo, en las que un agente es capaz de aprender tareas a partir del *feedback* que obtiene del entorno como consecuencia de sus acciones.

En particular, el reto consiste en entrenar un brazo robótico para que aprenda a realizar una serie de tareas de distinta complejidad, usando diferentes algoritmos de *Reinforcement Learning* (RL) y Deep Reinforcement Learning (DRL). Para ello, empleamos la herramienta S-RL Toolbox, la cual fue creada para estudiar el uso de estados de representación sobre algoritmos de RL. En este proyecto realizaremos los experimentos aplicando un aprendizaje de tipo *end-to-end*, donde el agente aprende directamente de las observaciones en crudo de los sensores (en nuestro caso, una cámara RGB).

En este trabajo ha sido esencial realizar una metodología que permitiese conocer mejor las características de los algoritmos que conforman el estado del arte de RL y DRL, y que son los utilizados en los experimentos. Gracias a esta investigación previa, ha sido posible extraer conclusiones razonadas y justificadas, comprendiendo mejor el comportamiento mostrado por cada algoritmo.

Cabe destacar que para que este tipo de aprendizaje obtenga buenos resultados son necesarias muchas repeticiones de la tarea hasta que el robot aprende "qué es lo que está bien y qué es lo que está mal". Es por ello que las pruebas se realizan en simulador, puesto que de este modo podemos acelerar la velocidad de cálculo y reducir riesgos experimentales.

Capítulo 2

Estado del arte

Los robots industriales nacieron como una evolución de la maquinaria automática convencional, con la ventaja de que son fácilmente adaptables a multitud de tareas de manipulación. Normalmente, los robots necesitan un técnico o ingeniero especializado que defina previamente cada una de las trayectoria que el robot debe seguir durante la tarea. Este hecho provoca que ante cualquier mínimo cambio en el proceso de producción, sea necesaria la reprogramación de las trayectorias.

Un robot se suele equipar con multitud de sensores de presencia, sensores de visión, etc junto a señales de comunicación que le permiten interactuar con su entorno. Sin embargo, sigue presente el mismo problema: normalmente el robot se comporta como un agente que solo reacciona a lo que se le ha predefinido. Por lo que hay una carencia importante de genericidad.

Por otro lado, en el campo de la investigación, la Inteligencia Artificial (IA) es una de las áreas con mayor desarrollo y potencial a día de hoy. Desde la aparición de AlexNet (Alex Krizevsky 2012) [1], el aprendizaje mediante redes neuronales profundas (*deep learning*) se aplica cada vez en más campos con la finalidad de descubrir soluciones alternativas y más efectivas que las establecidas actualmente.

En este apartado profundizaremos en la teoría de la inteligencia artificial y en particular en la del aprendizaje por refuerzo, con el objetivo de abrir el camino a aplicaciones robóticas que cuenten con este tipo de tecnología. Las técnicas de *Reinforcement Learning* (RL) están basadas en unos principios básicos que resaltaremos a lo largo de este proyecto. Conseguir un sistema capaz de adaptarse y aprender constantemente, que modifique su modelo automáticamente a medida que se va encontrando con nuevas situaciones, no

solo permitiría implantar la robótica en muchos más sectores de la industria, sino que también supondría un importante paso adelante para poder aplicar la robótica fuera de esta.

2.1. Introducción a la Inteligencia Artificial

En primer lugar, debemos esclarecer qué se entiende por Inteligencia Artificial (IA).

Un agente inteligente es aquel capaz de percibir su entorno y adaptarse, imitando funciones cognitivas propias de los seres humanos tales como el razonamiento y la solución de nuevos problemas. Dicho de otro modo, la inteligencia artificial es la capacidad de un sistema para interpretar correctamente datos externos, aprender de dichos datos y emplear esos conocimientos para lograr tareas y metas concretas a través de la adaptación [2].

En el caso del RL, el agente selecciona las acciones a realizar teniendo en cuenta un objetivo a corto plazo (que consistirá en maximizar sus posibilidades de éxito o recompensa), sin perder de vista el objetivo a largo plazo de resolver una tarea.

Los sistemas inteligentes combinan una memoria duradera, firmeza en sus metas, y una toma de decisiones basadas en su estado actual y estado objetivo que les permite lograr un comportamiento extremadamente eficiente, especialmente ante problemas complejos [3]. Por esta y más razones es por la que los sistemas de IA toman cada vez más protagonismo en nuestro día a día y son considerados esenciales en campos como economía, medicina, ingeniería e industria, etc.

2.2. Introducción al aprendizaje automático

El aprendizaje automático (*Machine Learning* en inglés) es una rama de la inteligencia artificial, cuyo objetivo es desarrollar técnicas que permitan que las computadoras aprendan [4]. Se dice que un agente aprende cuando mejora en la realización de una tarea a partir de datos de ejemplo o de la experiencia, o cuando adquiere habilidades que no poseía en sus inicios [5]. El aprendizaje automático resulta especialmente interesante en:

- Tareas en las que el algoritmo se debe adaptar a circunstancias particulares (hace falta un "entrenamiento"): detección de *spam*, reconocimiento del habla, etc
- Algoritmos difíciles de programar "a mano": reconocimiento del habla, de escritura

manuscrita, visión por computaddor, etc

Un factor clave en el que profundizaremos en los siguientes apartados es cómo conseguir que el sistema aprenda a comportarse de forma inteligente, dónde y cómo sentar las bases de dicha inteligencia. En el aprendizaje automático encontramos dos ramas que hacen referencia a dichas cuestiones [6]:

- Por un lado, algunos sistemas de aprendizaje automático intentan mantenerse independientes y no necesitar de la intuición o conocimiento experto. Es decir, no se intenta imponer al sistema una forma predeterminada de aprender, sino que el sistema parte desde cero hasta encontrar su método de aprendizaje (a lo que llamaremos política). En ocasiones esta política coincidirá con la humana pero en otras ocasiones no.
- Por otro lado, encontramos los sistemas que buscan establecer una colaboración entre el experto y la computador. A pesar de que esto pueda parecer más sencillo, en la mayoría de ocasiones no lo es. Debe tenerse muy en cuenta la complejidad que acarrea el tratar de imitar la intuición humana, ya que el diseñador del sistema ha de especificar la forma de representación de los datos y los métodos de manipulación y caracterización de los mismos.

Tanto si se elige un método u otro, al final con el aprendizaje automático siempre obtendremos un modelo que define el comportamiento que el agente ha aprendido para solucionar la tarea. Podemos comparar el modelo con "la solución del problema". Entre los modelos más comunes encontramos los geométricos, probabilísiticos, lógicos, de agrupamiento y de gradiente.

En cuanto a los algoritmos de aprendizaje automático, podemos agruparlos en cuatro grandes familias:

1. **Aprendizaje supervisado**. En estos algoritmos se hace uso de un conjunto de entrenamiento para adaptar el modelo de clasificación a unos datos concretos, con la finalidad de que sea capaz de asignar la clase de cada una de las muestras. Un clasificador supervisado consta de dos fases:

   a) Entrenamiento. Inicialmente se debe crear el modelo de entrenamiento a partir de muestras etiquetadas. Junto al conjunto de entrenamiento se utiliza el conjunto de validación, que se usa para verificar que el modelo generaliza bien y poder identificar cuándo terminar el entrenamiento. El objetivo es encontrar una generalización de la función a aprender que nos permita clasificar nuevos

ejemplos cuya clase es desconocida.

b) Reconocimiento o clasificación. Se usa una muestra que no se ha visto en la fase anterior, y se obtiene su clase tras aplicar el modelo a esa entrada.

Entre los métodos de aprendizaje supervisado más destacados encontramos las redes neuronales y convolucionales, árboles de decisión, Support Vector Machines y Naïve Bayes.

2. **Aprendizaje no supervisado**. En este caso no es necesario tener un conjunto de entrenamiento etiquetado. La idea es aprender mediante observación sin tener un conocimiento previo. A medida que se le proporcionen muestras no etiquetadas, el clasificador debe ser capaz de agruparlas según unas características comunes (*clustering*). Con los datos de cada *cluster* se construye un modelo que permitirá predecir a qué clase pertenecen los nuevos datos que queramos clasificar.

   Algunos de los métodos más destacados de aprendizaje no supervisado son: K-Means, KNN, Gaussian Mixture Models (GMM) y Expectation Maximization (EM).

3. **Aprendizaje semi-supervisado**. Este tercer tipo de aprendizaje reúne las características de los dos anteriores. Necesita un conjunto de entrenamiento amplio, en el que existe una pequeña proporción de muestras etiquetadas frente al resto que no están etiquetadas.

4. **Aprendizaje por refuerzo**: se trata de una técnica en la que no se utiliza un conjunto de datos para realizar el aprendizaje, sino que se aprende en base a la experiencia. En este caso, un agente interactúa con el entorno mediante acciones y modifica su modelo en función de la realimentación que obtiene de este. El objetivo del agente consiste en aprender qué secuencia de acciones le permiten maximizar su rendimiento. En los siguientes apartados profundizaremos en este tipo de aprendizaje, puesto que se trata del tema central del proyecto.

2.3. Redes Neuronales

Las redes neuronales son una rama del aprendizaje automático inspirado en las neuronas biológicas. Se trata de un sistema conexionista de neuronas que colaboran entre sí para producir una respuesta de salida. A estas conexiones se les asignan unos pesos que irán variando a medida que el sistema adquiere experiencia.

Las redes neuronales profundas (*Deep Neural Networks*) despegaron en 2012, cuando Alex Krizhevsky ganó la competición de ImageNet con el desarrollo de AlexNet [1], una red convolucional para el reconocimiento de objetos en imágenes. Estableció un hito consiguiendo un extraordinario nivel de acierto y de mejora con respecto a los otros métodos. A partir de entonces, el aprendizaje profundo (*Deep Learning*) empezó a aplicarse en multitud de ámbitos, mejorando en casi todos ellos los resultados conseguidos hasta ese momento. Donde más se destaca su uso es en el reconocimiento de imágenes.

En las técnicas *deep learning*, el modelo aprende automáticamente las mejores características que describen cada clase, lo que lleva generalmente a conseguir tasas de acierto mayores. Esto es una gran ventaja si lo comparamos con otros métodos de aprendizaje automático tradicional, donde debemos definir una serie de características para clasificar las instancias. Esto se debe a que es muy complicado saber cuáles son las características y descriptores óptimos.

Si bien, debemos considerar que una de las desventajas del aprendizaje profundo es que no tenemos control sobre lo que se aprende exactamente o cómo se aprende. El conocimiento se "reparte"por la red y muchas veces no se alcanza a comprender cómo se ha llegado al modelo.

2.4. Aprendizaje por Refuerzo (RL)

La inmensa complejidad de algunos fenómenos biológicos, políticos, sociológicos, etc hace que sea muy complicado razonar a partir de sus datos. La única forma de estudiarlos es a través de estadísticas, midiendo eventos superficiales e intentando establecer correlaciones entre ellos, incluso cuando no entendemos el mecanismo por el cual se relacionan. El aprendizaje por refuerzo (*Reinforcement Learning*, RL), y las redes neuronales profundas (*Deep Neural Networks, DNN*) son algunas de esas estrategias, y se basan en el muestreo para extraer información de los datos [7].

El aprendizaje por refuerzo ha ido evolucionando notablemente en los últimos años. Algunos de los ejemplos más fascinantes en el área de investigación son la aparición de las arquitecturas de DeepMind y Deep Q-learning de 2014, batiendo al entonces campeón del juego Go con AlphaGo en 2016, y la revolución de OpenAI y el algoritmo PPO en 2017, entre otros [8].

Esta estrategia de aprendizaje consiste en que un agente aprenda a comportarse en base a la experiencia (prueba y error) en un entorno, realizando acciones y viendo los

resultados, es decir, interactuando con este. Para conseguir esto es necesario una retroalimentación, a la que llamamos "recompensa", por las consecuencias de sus acciones (última acción o la secuencia de las últimas k acciones) que será obtenida por una función a la que llamaremos "función de recompensa".

En la mayoría de entornos existen ciertas restricciones a las acciones que el agente puede realizar. Llamaremos espacio de acciones al conjunto de acciones que el agente puede realizar en el entorno dado. Además, este conjunto será discreto o continuo en función del entorno. Cuando hablamos de entorno también estamos hablando de la tarea que debe realizar el agente. Por tanto, al igual que los entornos, las tareas pueden ser episódicas o continuas:

- Las tareas episódicas son aquellas en las que tenemos un estado inicial S_i y un estado final S_t (como es el caso que trataremos en este proyecto). La tarea se lleva a cabo a través de un conjunto de estados, acciones y recompensas.
- Las tareas continuas son aquellas que no tienen definidos un un inicio y un final. Son tareas que transcurren por siempre.

El modelo básico de aprendizaje por refuerzo se define como un proceso de decisión de Markov, y consiste en [9]:

1. Un conjunto de estados del agente en el entorno S.
2. Un conjunto de acciones A que el agente puede realizar
3. Reglas de transición entre los estados, siendo $P(s, s') = P_r(s_{t+1} = s'|s_t = s, a_t = a)$ la probabilidad de transición del estado s al estado s' aplicando la acción a.
4. Reglas que determinan la recompensa inmediata de una transición, siendo $R_a(s, s')$ la asociada a la transición del estado s al s' aplicando la acción a.
5. Reglas que describen lo que observa el agente y que nos ayudan a quedarnos con la información relevante del entorno.

El agente interactúa con el entorno durante una secuencia de instantes de tiempo t (a un instante de tiempo lo denominaremos "paso"). Para cada paso el agente recibe una representación del entorno, a la que llamaremos observación o_t. A partir de esa observación se considera el estado más probable en el que debe encontrarse el agente s_t, y seguidamente se escoge la acción a_t, dentro del conjunto de acciones posibles para el estado actual, que tiene mayor probabilidad de obtener una recompensa máxima. La acción repercutirá sobre el entorno, actualizando su estado a s_{t+1} y entregando una

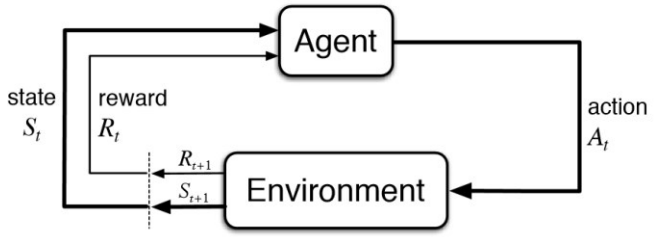

Figura 2.1: Interacción entre el agente y el entorno en RL [9]

recompensa numérica al agente [9].

Llamaremos trayectoria a una secuencia de estados y acciones en el entorno.

Una forma gráfica de resumir los conceptos anteriores es mediante el esquema de la figura 2.4, donde podemos apreciar que los dos componentes principales de un sistema de RL son el agente y el entorno en el que actúa. En la interacción entre estas dos partes aparece el concepto de observabilidad. Hablamos de plena observabilidad cuando suponemos que el agente posee conocimiento completo del estado actual del entorno. Y hablamos de observabilidad parcial cuando el conocimiento es incompleto.

La forma de aprender que define el RL está muy ligada a la forma natural de cómo los seres humanos interactuamos y aprendemos del medio. Esto puede verse con un ejemplo práctico muy sencillo:

Imagina que eres un niño en una sala de estar. Ves una chimenea y te acercas a ella. Es cálido, es positivo, te sientes bien (Recompensa positiva +1). Entiendes que el fuego es algo positivo. Pero luego intentas tocar el fuego. ¡Ay! Te quemas la mano (recompensa negativa -1). Acabas de comprender que el fuego es positivo cuando se encuentra a una distancia suficiente, ya que produce calor. Pero acércate demasiado y te quemarás. Así es como aprenden los humanos, a través de la interacción. El aprendizaje por refuerzo es solo un enfoque computacional para aprender de la acción [8].

Un aspecto ya mencionado anteriormente y en el que vale la pena detenerse es la función de recompensa. Está función es la encargada de asignar la información de recompensa correspondiente (positiva o negativa) asociada a alcanzar ciertos estados (pueden ser todos). El agente recibe dicha información y aprende a medida que se van reforzando aquellas decisiones que nos llevan a recompensas mayores y debilitando las que nos llevan a recompensas menores.

El objetivo principal es aprender la función de recompensa que ayude al agente inte-

ligente a maximizar la recompensa acumulada esperada, y así optimizar su modelo para comprender el comportamiento del entorno y tomar buenas decisiones para el logro de sus metas [9].

Obtener una buena función de recompensa es un aspecto clave para el éxito del aprendizaje puesto que el principal objetivo del agente es maximizar la recompensa acumulada esperada, dado que en RL todos los objetivos se pueden describir mediante la maximización de la recompensa acumulada esperada.

Otro aspecto importante a tener en cuenta es que el agente debe razonar sobre las consecuencias de sus acciones a corto y largo plazo para poder actuar de manera óptima. Es decir, es posible que el agente pueda realizar una acción que le proporcione una inmensa recompensa inmediata pero que no sea viable a largo plazo. Y por el contrario, es posible que la acción siguiente no suponga una gran recompensa inmediata, pero sí un inicio al camino de una gran recompensa a largo plazo. A este dilema se le conoce comúnmente en inglés como *exploration/exploitation trade-off*. La clave consiste en definir una regla que nos ayude a tomar la mejor decisión en cada caso.

Podemos formular la recompensa acumulada esperada en cada instante de tiempo t como [8] :

$$G_t = R_{t+1} + R_{t+2} + ...$$

lo que es equivalente a:

$$G_t = \sum_{k=0}^{T} R_{t+k+1}$$

Sin embargo, en la realidad no podemos sumar las recompensas de este modo. Las recompensas más cercanas al estado inicial de la ejecución de la acción tienen mayor probabilidad de ser alcanzadas que las recompensas a largo plazo. Como consecuencia obtenemos que no todas las recompensas valen lo mismo, están afectadas con el paso del tiempo, el estado actual, los estados anteriores, etc. Para lidiar con esto y plasmar el concepto de exploración/explotación, se introduce un factor de descuento γ que varía entre 0 y 1, y que está asociado con la probabilidad de obtener la recompensa. Cuanto menor sea γ, mayor será el descuento aplicado sobre la recompensa. Esto significa que el agente se preocupa más por la recompensa a largo plazo. Por el contrario, cuanto mayor sea γ, menor será el descuento, es decir, nuestro agente se preocupa más por la recompensa a corto plazo [8].Esto es muy interesante, puesto que al inicio del entrenamiento nos interesa un valor de γ mínimo, para que le agente se arriesgue a explorar todo lo

posible y aprenda más. A medida que el agente vaya aprendiendo iremos aumentando el valor de γ.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} where \gamma \in [0, 1)$$
$$= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} ...$$

En función de cuando se recogen las recompensas obtenidas, podemos clasificar los algoritmos de RL en dos métodos [9]:

- En el método de **Monte Carlo**, el agente tiene en cuenta la recompensa obtenida solo al final de cada episodio. En dicho momento, revisará cómo de bien se ha realizado la acción y modificará el modelo y la máxima recompensa esperada en consecuencia.
- En el método de **Temporal Difference Learning**(TD), el agente tiene en cuenta la recompensa tras cada instante de tiempo, momento en el que se actualizará su modelo y se seleccionara la siguiente acción como aquella que maximice la recompensa esperada.

Una vez definidas las bases del aprendizaje por refuerzo y las dos formas principales de aprendizaje en dicho campo, ahora veremos las diferentes aproximaciones para la resolución de problemas mediante RL, puesto que, aunque el tema de la exploración se tiene en cuenta, el problema sigue siendo conocer qué acciones son las adecuadas en consideración con la experiencia pasada.

1. **Aproximación basada en Valor**. En este caso el objetivo consiste en optimizar una función que representa la recompensa máxima esperada que el agente puede obtener en cada instante de tiempo, a esta función la llamaremos *función de valor* y se representa como $V(s)$. Dicho de otro modo, el V de un estado es la recompensa total que un agente podría llegar a acumular en el futuro partiendo de dicho estado. Gracias a esto, el agente es capaz de tomar la mejor acción para cada estado en el que se encuentre, que será aquella que proporcione el mayor valor [8]. Esta definición coincide con la de función de recompensa, y es que todos los algoritmos de RL cuentan con ella, pero no todos la utilizan del mismo modo. Este método funciona especialmente bien cuando se tiene un espacio de estados y un espacio de acciones finito. Si bien, una de sus principales desventajas es que puede generar grandes oscilaciones durante el entrenamiento. Esto normalmente sucede debido a

que la elección de cualquier acción puede cambiar dramáticamente por un pequeño cambio arbitrario en los valores estimados por la función de valor [10].

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s \right]$$

Expected — Reward discounted — Given that state

Figura 2.2: Definición genérica de la función de valor [10]

2. **Aproximación basada en Política**. La política (*policy*) define el comportamiento del agente en cada estado, es decir, será lo que determine qué acción se toma en cada instante en función del estado actual. En este caso la política $\pi(s)$ se optimiza directamente, sin preocuparse por la función de valor. Si bien, en algunos casos si se emplea una función de valor para optimizar los parámetros de la política, pero no será usada para seleccionar la acción a tomar por el agente [10].

$$a = \pi(s)$$

Los métodos basados en política se dividen a su vez en dos grupos: por un lado tenemos a las políticas deterministas, que para el mismo estado siempre devuelven la misma acción y que normalmente son empleadas en entornos deterministas. Y por otro lado tenemos las políticas estocásticas, cuyo resultado es una distribución de probabilidad del conjunto de acciones posibles para cada estado y que normalmente empleadas cuando el agente se encuentra en un entorno desconocido [11].

La principal complejidad de este método reside en encontrar una función que evalúe correctamente que tan buena es la política empleada.

3. **Aproximación basada en Modelo**, donde el entorno es un modelo que nosotros podemos modificar. El inconveniente de este caso es que son necesarias diferentes representaciones del modelo para cada entorno. No profundizamos más en esta aproximación ya que en este proyecto no trataremos ningún algoritmo de este tipo, además de que son más complejos de entender.

2.5. Deep Reinforcement Learning

El RL convencional resulta ineficiente en entornos donde el espacio de estados y el conjunto de acciones posibles son muy grandes. Esto se debe a que el agente debe hacer una gran exploración hasta tener suficiente conocimiento como para obtener un resultado razonable. Por ejemplo, en el juego del ajedrez, el espacio de estados está limitado al número de casillas del tablero y fichas de cada jugador, lo cual resulta bastante asequible. Sin embargo, ahora pongamos el caso de la tarea de este proyecto, un entorno 3D donde el espacio de estados está definido por todas las posibles configuraciones que un brazo robótico sea capaz de alcanzar, mas todas las posibles posiciones del objetivo. Evidentemente, en este segundo caso el número de estados y acciones es significativamente superior.

Como solución a este problema aparece el aprendizaje profundo por refuerzo (*Deep Reinforcement Learning*), donde se emplean redes neuronales profundas (*Deep Neural Networks*) para solucionar las limitaciones del aprendizaje por refuerzo. En este caso, para cada acción posible en el estado actual, la red neuronal se encarga de devolver la aproximación de la recompensa acumulada esperada correspondiente. Durante el entrenamiento, la red debe encontrar los coeficientes que mejor aproximen la función que relaciona la entrada con la salida del sistema, a través de una ajuste iterativo [12].

Al comienzo del aprendizaje por refuerzo los coeficientes de la red neuronal se inicializan estocástica o aleatoriamente. Usando la retroalimentación del entorno, la red neuronal puede usar la diferencia entre su recompensa esperada y la recompensa del *ground-truth* para ajustar sus pesos y mejorar su interpretación de las tuplas de estado-acción. Este bucle de retroalimentación es análogo al proceso de *backpropagation* en el aprendizaje supervisado. Si bien, debe tenerse en cuenta que las recompensas devueltas por el entorno pueden variar, retrasarse o verse afectadas por variables desconocidas, lo que puede introducir ruido en el bucle de retroalimentación [7].

Asimismo, las redes convolucionales pueden utilizarse en RL para reconocer el estado del agente y del objetivo. Es decir, nos permiten transformar información en crudo en información útil para el agente. Por ejemplo, a partir de una imagen 2D, que la red aprenda a identificar cual es la posición del agente y del objetivo. Concretamente, el principal objetivo de esta alternativa es reducir la complejidad de los estados para reducir de este modo el tiempo de computación necesario para el entrenamiento.

A continuación se muestra la analogía entre un clasificador basado en redes convolu-

cionales y un agente convolucional:

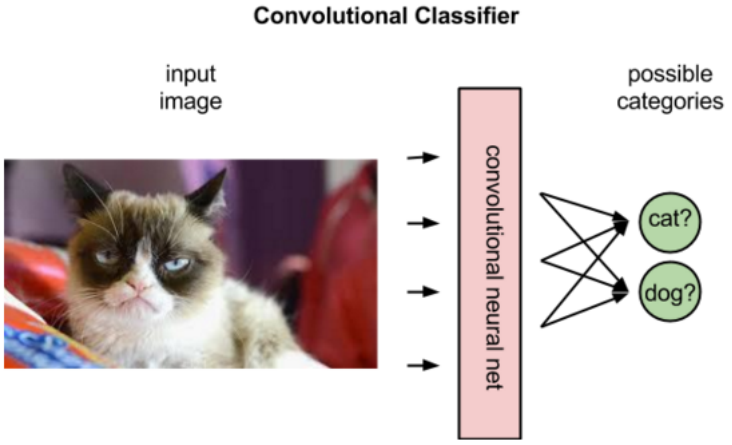

Figura 2.3: Clasificador convolucional [7]

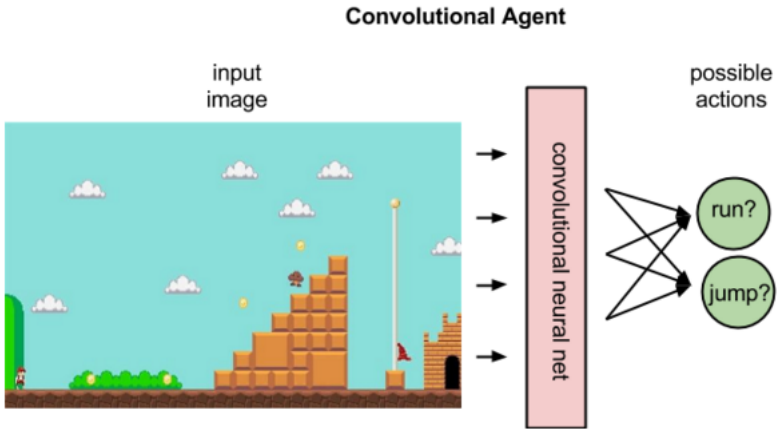

Figura 2.4: Agente convolucional [7]

Capítulo 3

Tecnologías empleadas

3.1. S-RL Toolbox

S-RL Toolbox [13] es un repositorio creado para el análisis de métodos de aprendizaje de representación de estado (*State Representation Learning*) aplicado a RL. Nosotros haremos uso de este repositorio con la finalidad de comprender el funcionamiento de RL y Deep RL y ser capaces de aplicar diversos algoritmos de esta metodología que nos permitan resolver problemas de distinta complejidad. Este procedimiento nos ayudará a entender los comportamientos, explicarlos, y extraer conclusiones. S-RL Toolbox nos proporciona herramientas para entrenar el sistema, guardar el modelo del agente entrenado, cargar el agente y visualizar resultados, etc. Esta librería también incluye una serie de entornos Gym donde simular los entrenamientos y poder analizar los resultados del aprendizaje de un modo visual.

3.2. Anaconda

Para poder hacer uso de S-RL Toolbox hemos instalado Anaconda en nuestra máquina de trabajo. Anaconda es una distribución libre y abierta del lenguajes R y Python, utilizada en ciencia de datos, y aprendizaje automático, lo que nos permite realizar procesamientos de grandes volúmenes de información, análisis predictivos y cálculos científicos [14] típicos de aprendizaje automático.

Las tres grandes ventajas que han hecho que nos decidamos por trabajar con esta herramienta son:

1. Nos permite instalar todas las dependencias de una sola vez y de forma cómoda, a través de un fichero de entorno que llamaremos `environment.yml`. En este fichero, incluiremos todos los paquetes y dependencias necesarias en Python para poder trabajar con el repositorio.
2. Podemos crear entornos vírgenes con los que trabajar con Python libremente, con independencia de lo ya instalado en el equipo de trabajo, y sin miedo de ocasionar incompatibilidades en nuestro equipo.
3. Facilita trabajar con cualquier versión de Python, independientemente de la que tengamos instalada en nuestra máquina.

3.3. Python

El código utilizado e implementado en este trabajo está escrito en lenguaje Pytnon.

Python es un lenguaje de programación que nació en 1991 por la mano de Guido van Rossum. Especialmente en la última década, se ha convertido en uno de los lenguajes más populares gracias a su gran portabilidad (ya sea en Mac, Linux o Windows), esto se debe gracias a que la mayoría de las librerías más comunes ya están dentro del intérprete. Y las que no lo están, se instalan con facilidad gracias a la herramienta **pip**.

Se trata de un lenguaje de programación caracterizado por ser multiparadigma. Esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional. Además, otros paradigmas pueden ser soportados mediante el uso de extensiones [15].

Otro de los aspectos importantes que nos aporta Python y que nos hizo decidirnos por él para este trabajo, es la facilidad de extensión que nos proporciona, es decir, nos permite escribir nuevos módulos fácilmente en otros lenguajes como C o C++, lo que es una gran ventaja debido a que estos dos últimos son los lenguajes sobre los que más hemos trabajado a lo largo del grado de Ingeniería Robótica.

Además, Python también facilita el trabajo con Tensorflow, una librería fundamental para el desarrollo de este proyecto y que explicaremos en el siguiente punto.

Si bien, la simplicidad que nos proporciona este lenguaje acarrea la desventaja de no ser un lenguaje tan óptimo como otros lenguajes de alto nivel como por ejemplo C y C++. Algo que ha afectado significativamente en los tiempos de ejecución de los

programas que aquí se estudian.

La versión empleada para la realización de los experimentos es Python 3.5.

3.4. Tensorflow

TensorFlow es una herramienta de cálculo computacional creada por Google, destinada principalmente para su uso en aprendizaje automático. Actualmente es una plataforma de código abierto, por lo que se usa para cientos de proyectos tanto dentro como fuera de Google.

Se trata de la principal herramienta utilizada para implementar los algoritmos de RL en la librería de S-RL Toolbox, y a su vez de Stable Baselines.

TensorFlow [16] tiene un excelente equilibrio de flexibilidad y escalabilidad. La flexibilidad permite a los desarrolladores e investigadores probar ideas nuevas en un tiempo corto. Aunque muchos piensan que TensorFlow sólo sirve para redes neuronales profundas, realmente permite hacer todo tipo de tareas como aprendizaje por refuerzo (que es el principal interés de este proyecto).

TensorFlow, además, es portable, por lo que puede ser utilizado en todo tipo de dispositivos [16]. Este último aspecto nos deja una puerta abierta para implementar la solución que aquí se estudia en dispositivos de aplicación real, como podría ser la Rasberry Pi.

Todas estas ventajas descritas en el párrafo anterior hacen de Tensorflow la herramienta idónea para ser utilizada en este trabajo. Además, tiene la ventaja de ser especialmente sencilla tanto para principiantes como para experimentados, lo cual es otro punto a favor, debido a que es la primera vez que el alumno se enfrenta a un problema de este calibre.

3.5. Stable Baselines

Stable Baselines es una bifurcación de OpenAI que tiene como objetivo hacer que el aprendizaje por refuerzo sea accesible a más desarrolladores. Mediante OpenAI obtenemos algoritmos de RL que funcionan correctamente, si bien, a pesar de ser *open source*, la implementación está muy desorganizada y poco documentada, por lo que resulta muy complejo adaptarla a problemas concretos [17].

Stable Baselines da solución a los problemas de OpenAI mediante una presentación

más organizada, simplificada y unificada de los algoritmos, siguiendo una estructura común en sus clases y métodos. Todo esto facilita al sector de investigación e industria poder replicar, refinar y crear nuevas ideas para desarrollar proyectos de vanguardia. También abre las puertas a jóvenes estudiantes investigadores, como es el caso del autor de esta memoria.

Esta librería permite el entrenamiento de RL en funciones arbitrarias, es decir, los algoritmos de RL se pueden entrenar con algo más que píxeles (las actuales líneas base de OpenAI solo admiten acciones continuas cuando no se usan imágenes como entrada). Además, está mejor optimizado, lo que se ve reflejado significativamente en su rendimiento [17].

Por lo que a nuestro trabajo respecta, Stable Baselines viene incorporado en el repositorio de S-RL Toolbox, y nos proporciona la implementación de los algoritmos de RL y Deep RL que emplearemos en este trabajo. Para el correcto funcionamiento de los algoritmos ha sido indispensable la herramienta de TensorFlow, descrita en el apartado anterior.

3.6. Gym

S-RL Toolbox nos proporciona una serie de entornos personalizados en Gym para probar los algoritmos de RL en simulación.

Gym es una herramienta creada junto a OpenAI. Se trata de un simulador que fue desarroyado con el objetivo de proporcionar un punto de referencia de inteligencia artificial, fácil de instalar con una amplia variedad de entornos diferentes (algo parecido, pero más amplio, al reto de reconocimiento visual de gran escala ImageNet utilizado en la investigación de aprendizaje supervisado), y que espera estandarizar la forma en que se definen los entornos en las publicaciones de investigación de IA, de modo que la investigación publicada se vuelva más fácil de reproducir [18].

A efectos prácticos, Gym nos permite realizar un mejor análisis de cómo se comportan los distintos algoritmos de RL que proporciona Stable Baselines-OpenAI en simulación de problemas reales. Por lo que a este trabajo respecta, utilizaremos 4 entornos de simulación editados por S-RL Toolbox, en los que un robot antropomórfico Kuka modelo LBR iiwa deberá realizar distintas tareas. Estos entornos serán descritos con detalle más adelante.

La simulación es una parte crucial de cualquier proyecto, sobre todo en campos tan

innovadores como es la robótica, donde los materiales y equipos son muy caros. A esto se suma la peligrosidad que acarrea el manejo de máquinas con un alto alcance, grado de movimiento y velocidad, que puede producir consecuencias graves para las personas que estén trabajando cerca. Todo esto supone un factor de riesgo que se puede reducir enormemente al trabajar previamente en simulación.

Hoy en día los simuladores son cada vez más complejos y prácticamente consiguen replicar la realidad. Por tanto, no solo sirve como prevención de riesgos, sino también para ahorrar mucho dinero en investigaciones, evitando la compra de equipos antes de saber si la solución diseñada será posible de llevar a cabo o no. Si bien, el éxito en simulación nunca garantiza el éxito del proyecto en la realidad.

Capítulo 4

Metodología - Algoritmos de RL

En este apartado se pretende dar un conocimiento básico al lector para que sea capaz de entender y diferenciar los algoritmos de *Reinforcement Learning* (RL) con los que se experimenta en este proyecto.

4.1. Q-Learning

Q-Learning es un algoritmo de RL basado en valor. La idea principal de este algoritmo es crear una tabla, denominada tabla-Q, donde se representa la recompensa máxima esperada para cada posible acción en el espacio de estados del entorno. Por lo tanto, se trata de una tabla bidimensional (espacio de estados x espacio de acciones). Inicialmente la tabla partirá con valores aleatorios y esta se irá actualizando a medida que el agente interactúe con el entorno [19].

Gracias a la tabla-Q, para un estado dado el agente conoce cual es la acción que debe tomar para maximizar la recompensa futura. Es decir, el valor de cada elemento de la tabla-Q será la recompensa máxima esperada si se toma una acción concreta en un estado concreto. Esto ayuda al agente a decidir siempre la mejor acción.

Junto a la tabla-Q aparece la función-Q. Esta función proporcionará la recompensa esperada futura para un estado y una acción dados. Puede entenderse esta función simplemente como un lector de la tabla-Q.

La función-Q normalmente cuenta con un parámetro que indica en qué medida deben de aleatorizarse las decisiones del agente. Puesto que al principio nos interesa recorrer al máximo el espacio de estados, las acciones deben tomarse de una manera más aleatoria,

y a medida que el agente va aprendiendo (actualiza la tabla en función de la recompensa que recibe del entorno), interesará que las decisiones sean menos aleatorias para que el agente deje de explorar y optimice su comportamiento.

El procedimiento de aprendizaje es muy sencillo y puede resumirse en el siguiente esquema:

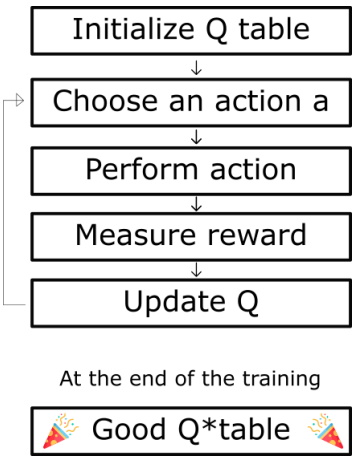

Figura 4.1: Proceso de aprendizaje de Q-learning [19]

4.2. Deep Q-Learning (DQL)

Una vez comentado Q-Learning, es muy fácil comprender DQL. No es más que el mismo concepto pero sustituyendo la Q-table y la Q-function por una red neuronal. A esta red le proporcionaremos el estado actual en cada instante de tiempo y ella nos dará una aproximación de cual es la recompensa máxima que podemos alcanzar para cada acción posible en dicho estado. Del mismo modo, elegiremos la acción que maximice la recompensa futura.

Emplear una red neuronal en vez de una tabla tiene grandes ventajas:

- Puede usarse en entornos donde el espacio de estados es grande. En el caso de Q-learning, definir y actualizar la tabla-Q resulta ineficiente y costoso computacionalmente en entornos con un gran espacio de estados.
- A pesar de que Q-learning es un algoritmo muy potente, su principal debilidad es la falta de genericidad. En Q-learning, el agente no es capaz de estimar qué acción debe tomar en los estados que todavía no ha visitado. DQN aparta este

problema sustituyendo la matriz bidimensional por una Red Neuronal para estimar la Función de valor [20].

El algoritmo Q-learning no se aplicará en los experimentos de este proyecto dadas sus limitaciones. En este caso se ha introducido para comprender mejor el algoritmo DQL.

Antes de explicar los siguientes algoritmos, también es importante comprender dos sub-métodos de RL derivados de las aproximaciones basadas en política y/o valor. Se trata de *Policy Gradient* y *Actor-Critic*. Ambos están a la vanguardia del RL y la mayoría de algoritmos que veremos están basados al menos en uno de ellos, y que surgen con el objetivo de solucionar sus desventajas.

4.3. Policy Gradient (PG)

Policy Gradient (PG) es un submétodo derivado de la aproximación basada en política. Su objetivo consiste en actualizar la política en la dirección de máximo crecimiento de la recompensa. Esto se consigue a través de la denominada *Policy Gradient Objective Function*, que por cada paso de ascenso de gradiente guía al agente a tomar acciones que conduzcan a mayores recompensas.

Para que este método resulte eficiente se utiliza una derivada de primer orden y se aproxima la superficie del espacio de estados para que sea plana. Esto conlleva un alto riesgo, puesto que si la superficie original tiene una curvatura alta, podemos hacer movimientos que conduzcan a desaprender bruscamente. Es decir, un cambio demasiado grande en la política puede conducir al desastre pero, al mismo tiempo, debemos considerar que si el cambio es demasiado pequeño, el modelo aprenderá muy lento. Por lo tanto, una de las principales dificultades de este método consiste en ajustar dicho parámetro [21].

Otras desventajas de este método son:

- Resulta complicado mapear los cambios de política en los parámetros de esta.
- Tiene una eficiencia de muestreo pobre, necesitando de millones de pasos de entrenamiento para experimentos donde el agente actúa sobre un entorno de tres dimensiones. Por lo que para la vida real, en aplicaciones de robótica supone un costo computacional demasiado elevado [21].

Como solución a los problemas de PG nacen los algoritmos de TRPO y PPO (que explicaremos más adelante), donde se busca limitar los cambios en las políticas que

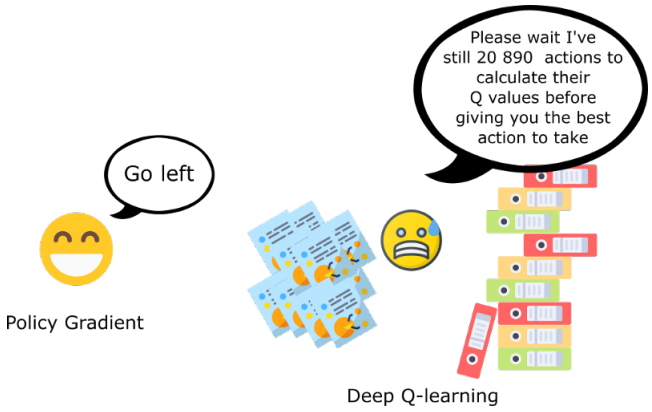

Figura 4.2: Policy Gradient vs Value based method

se están aprendiendo, e idealmente, que cualquier cambio garantice una mejora en las recompensas.

En cuanto a las ventajas de este método destacamos los siguientes puntos:

- Es un método basado en política y estos tienen mejores propiedades de convergencia.
- Puede utilizarse para aprender políticas estocásticas (lo que no pueden hacer los algoritmos basados en valor).
- Obtiene mejores resultados en espacios de acción de alta dimensionalidad (cómo es el caso de este proyecto), o cuando se usan acciones continuas.

4.4. Actor-Critic (AC)

Actor-Critic es un método híbrido entre el método basado en valor y el basado en política, tratando de unir las ventajas de ambos.

El método Actor-Critic consta de dos partes, normalmente aproximadas mediante redes neuronales [11]:

- *Critic*: se encarga de evaluar cómo de buena es la acción tomada. Es la parte que copia el modelo basado en valor.
- *Actor*: toma las decisiones en función de la información que le proporciona *Critic*, es decir, se encarga de controlar el comportamiento del agente. Sigue el modelo basado en política.

Ambas redes trabajan en paralelo, lo que significa que tenemos dos grupos de pesos que deben ser optimizados separadamente. Sin embargo, su entrenamiento es dependiente.

Al principio del entrenamiento, puesto que el *Actor* no tiene ningún conocimiento de cómo debe comportarse, realizará acciones de forma aleatoria. Ante estas acciones, el *Critic* se encargará de comunicarle si las acciones tomadas son buenas o malas, y el *Actor* actualizará su política de toma de decisiones en consecuencia. De la misma forma, el *Critic* también actualizará su manera de evaluar las acciones y enviarle información al *Actor*. De este modo es como el feedback entre ambos mejora a lo largo del entrenamiento [22].

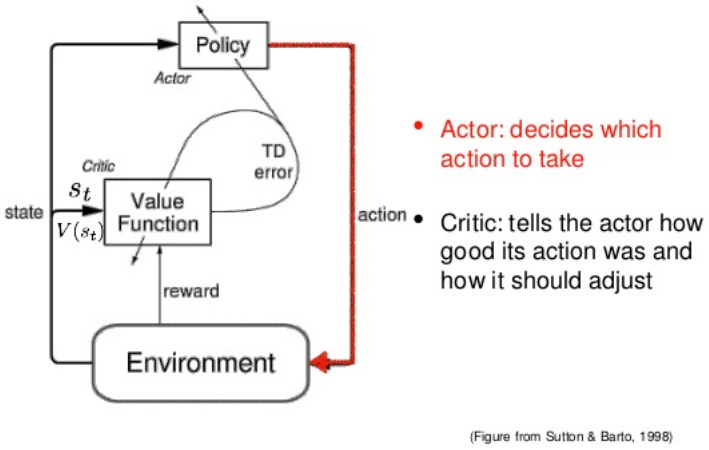

Figura 4.3: Proceso de aprendizaje del método Actor-Critic [23]

Entender la arquitectura *Actor-Critic* nos será muy útil para comprender el estado del arte de algoritmos basados en dicho método, como son A2C y PPO.

4.5. Advantage Actor Critic (A2C)

Advantage Actor-Critic (A2C) es un algoritmo de RL que sigue la estructura *Actor-Critic*. Surgió como una versión síncrona y determinista de A3C (*Asyncronous Advantage Actor-Critic*) tras ver que este último no proporcionaba ventajas claras. Además, una característica importante de A2C frente A3C es que es más eficiente en GPU, por lo que ofrece mayor rendimiento ante espacios de acción de alta dimensionalidad [23].

Para poder explicar el algoritmo A2C antes debemos definir qué es la Función de ventaja (*Advantage Function*). Esta función sustituye a la función de valor de la aproxi-

mación basada en valor, con el objetivo de reducir la alta variabilidad que esta acarrea, obteniendo así un modelo más estable. *Advantage* es un término que aparece comúnmente en gran cantidad de algoritmos de RL. Lo podemos entender como: cómo de buena es una acción comparada con el resto para un estado específico. Por lo tanto, el *Critic* aprenderá los *Advantage values*.

La Función de ventaja se define como:

$$A(s, a) = Q(s, a) - V(s)$$

donde $Q(s, a)$ es el valor que proporciona la función de valor para el estado y la acción actual, $V(s)$ es la media de los valores de recompensa proporcionados para ese estado, y $A(s, a)$ es la recompensa adicional recibida para una acción concreta para un estado dado. La recompensa extra es aquella que excede el valor esperado para ese estado.

La variación de la política se produce a partir del resultado que nos proporcione la función *Advantage*. Si el resultado de esta es mayor que 0, quiere decir que la acción estudiada es mejor que la media para el estado actual. Por lo tanto, el objetivo es emplear esta función para incrementar la probabilidad de tomar esa acción en dicho estado. Por el contrario, cuando la función *Advantage* da un resultado menor que cero, la política se actualizará para dejar de tomar decisiones con la tendencia que lo estaba haciendo [24] .

En A2C se lanzan m copias diferentes del mismo entorno en paralelo (de este modo, es posible explorar una mayor parte del espacio estados en un tiempo menor), donde una primera red interactuará con todos ellos durante n instantes de tiempo, lo que proporcionará una pila de conocimiento. Con dicho conocimiento se entrenará la segunda red, que representa el modelo de entrenamiento, y cuyos pesos se actualizarán de forma síncrona, solo cuando se haya recogido el resultado de todos los entornos lanzados con la primera red. Finalmente, a partir de el modelo de entrenamiento actualizado, se vuelven a lanzar m entornos en paralelo, para volver a repetir el proceso explicado [23].

4.6. Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization (TRPO) es un algoritmo derivado principalmente de PG. Además, parte de su estructura también deriva de *Actor-Critic*, pero modificando la forma en que se actualizan los pesos del Actor [25].

Tal y como se ha explicado en su descripción, el principal inconveniente de PG es el tamaño del paso. Donde un paso demasiado grande puede llevar bruscamente a un gran desaprendizaje. Y con un paso demasiado pequeño el aprendizaje se eterniza. TRPO es uno de los algoritmos más reconocidos por solucionar este problema.

Al igual que PG, TRPO es un algoritmo basado en política donde esta se actualiza seleccionando en cada instante de tiempo la acción que proporcione el mayor aprendizaje posible en el comportamiento del agente (ascenso de gradiente). Si bien, a diferencia del PG, TRPO solo se comporta de este modo siempre y cuando se satisfaga una restricción especial que controla cómo de parecidos son los comportamientos de las políticas anterior (antes de la última actualización) y actual [26].

Dicho de otra forma, la mayoría de algoritmos basados en PG intentan mantener las políticas anterior y actual cerca en el espacio de parámetros, pero esto no es efectivo porque pequeños cambios en los parámetros pueden llevar a grandes cambios en el comportamiento de la política. TRPO soluciona esto olvidándose de cómo de parecidos son los parámetros y centrándose en que el comportamiento del agente no varíe bruscamente.

Entre las principales ventajas de esta algoritmo destacamos:

- Es un método basado en política por lo que tiene mejores propiedades de convergencia.
- Presenta buen comportamiento tanto en entornos con espacio de acción continuo como discreto.
- Nos permite despreocuparnos más del tamaño del paso.
- Según los experimentos llevados a cabo en [25], TRPO proporciona normalmente una mejora monotónica en el aprendizaje, sin ser necesario un gran ajuste de sus hiperparámetros.
- Nos permite optimizar de forma efectiva complejas políticas no-lineales, como serían aquellas basadas en redes neuronales [25].

Por otro lado, en cuanto a las ventajas señalamos los siguientes puntos:

- Para solucionar el problema de PG lleva a cabo un complejo método/optimización de segundo orden (por lo que es un algoritmo más complejo y con más carga computacional que otras alternativas del estado del arte).
- En la misma página web de OpenAI podemos encontrar la siguiente advertencia: "TRPO, a pesar de ser útil en tareas de control continuo, no es fácilmente compa-

tible con algoritmos que comparten parámetros entre una política y una función de valor o pérdidas auxiliares, como los utilizados para resolver problemas en dominios donde la entrada visual es significativa"[27]. Este es un factor importante a tener en cuenta en nuestros experimentos y que trataremos de comprobar, debido a que la tarea a resolver que aquí se plantea depende principalmente de la entrada visual.

4.7. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) es un algoritmo bastante reciente (2017-2018), que no deja de ser uno más de los que surge intentando solucionar los problemas de PG. Al igual que TRPO, PPO surge con la motivación de conseguir que cada paso garantice una mejora máxima en la política, es decir, conseguir un muestreo más eficiente. Mientras TRPO trata de conseguir dicho objetivo utilizando un complejo método de segundo orden, PPO pertenece al grupo de algoritmos que trata de hacerlo mediante optimización de primer orden, junto a otras técnicas sencillas [28].

PPO no actúa sobre la función objetivo restringiendo las actualizaciones de política que producen un cambio demasiado grande en el comportamiento de esta, sino que las penaliza [24]. En el artículo donde se publicó PPO podemos ver algunos de los métodos más comunes para implementar este concepto [28].

Otra característica importante de PPO es que también emplea la función de ventaja explicada en A2C. Sin embargo, en este caso queremos penalizar aquellas acciones que nos conduzcan hacia una actualización brusca, por lo que, ante un resultado positivo de la función de ventaja incrementaremos la probabilidad de tomar la acción estudiada pero no demasiado.

PPO obtiene resultados que son comparables a los de otras aproximaciones que pertenecen al estado el arte de RL y que son mucho más complejas y computacionalmente costosas. Gracias a estas características, PPO se ha convertido en el algoritmo RL de referencia para OpenAI. De hecho, en los experimentos realizados por OpenAI en tareas de control continuo, PPO ha obtenido resultados muy cercanos a algoritmos especializados en este campo como ACER, a pesar de ser mucho más simple [27].

Podemos resumir las ventajas de este algorimo en los siguientes puntos:

- Se caracteriza por ser un algoritmo con una implementación más sencilla y con menor coste computacional, proporcionando resultados que compiten con algoritmos

del estado del arte con mayor complejidad [29].

- Aporta genericidad, obteniendo buenas soluciones para gran diversidad de problemas, tanto por el tipo de entorno como por la complejidad de la tarea. Además, estos buenos resultados los proporciona sin a penas tener que ajustar sus hiperparámetros [30].
- Puede ser usado tanto para entornos con espacio de acciones continuo como discreto.

En este proyecto emplearemos una implementación especial de PPO. Se trata de PPO2, una versión publicada por OpenAI que permite ser ejecutada en GPU, proporcionando un tiempo de ejecución tres veces menor que la implementación original, según declara la propia OpenAI [27].

4.8. Actor-Critic Experience Replay (ACER)

ACER es otro de los algoritmos de la familia de PG, que aplica la arquitectura Actor-Critic con repetición de experiencia, *Experience Replay* en inglés. En muchos casos es conocido como la versión *off-policy* de A3C. La repetición de experiencia le permite realizar más de una actualización de gradiente por cada paso [27]

La mayor dificultad de conseguir una versión *off-policy* de A3C es cómo controlar su estabilidad [31]. ACER propone tres principios para solucionarlo:

- Aplicar un mayor control sobre los pesos, truncándolos y aplicando una corrección de sesgo si fuese necesario. Esto ayuda a reducir la alta varianza generada por PG
- Copia la idea de TRPO pero con la intención de hacerlo computacionalmente más eficiente. En lugar de calcular la diferencia entre políticas antes y después de ser actualizada, ACER calcula un promedio de las políticas pasadas y fuerza que la política actualizada no se desvíe demasiado de dicho promedio.
- Emplea una estimación del valor Q que ayuda a garantizar la convergencia en casi cualquier objetivo propuesto.

Estas técnicas, junto al uso de la repetición de experiencia, no son algo sencillo y convierten a ACER en un algoritmo más complejo que PPO. Si bien, según publica OpenAI en su página web, ACER solo proporciona mejores resultados que en PPO en entornos concretos, [27].

4.9. Augmented Random Search (ARS)

Augmented Random Search (ARS) es un algoritmo basado en política y con características muy distintas a los ya explicados en este proyecto. ARS trata de resolver problemas de control continuo mediante un interesante método de búsqueda aleatoria para entrenar la política, basado en el artículo *Simple random search provides a competitive approach to reinforcement learning* [32].

Para actualizar los pesos de manera efectiva, lo que se hace en este algoritmo es obtener una matriz de valores pequeños aleatorios, y a continuación se le añade dicha matriz a la matriz de pesos del agente (red neuronal). Más tarde, en vez de sumar los valores lo que se hace es restarlos. Y luego se repite este mismo procedimiento muchas veces. El resultado es un agente que intenta realizar una tarea con pesos ligeramente diferentes en cada episodio. Como consecuencia, se obtienen diferentes recompensas por cada configuración de pesos generada al interaccionar con el entorno, unas recompensas serán más altas que otras. Lo que ARS hace es ajustar los pesos en función de la configuración de pesos que ha proporcionado una recompensa mayor. Se continuará ajustando dichos pesos hasta llegar a un máximo local o global. Aquellas configuraciones de pesos que den recompensas muy pobres se descartarán directamente, lo que ayuda a ahorrar tiempo y coste computacional. A este procedimiento se le denomina "método de diferencias finitas"[33].

A diferencia del resto de algoritmos vistos hasta el momento, ARS explora el espacio de políticas en vez del espacio de acciones. Esto significa que en vez de analizar la recompensa obtenida tras cada acción, se analiza la recompensa tras una serie de acciones con la finalidad de determinar si los pesos de la política utilizados en dichas acciones han conducido hacia una mayor recompensa [34].

Otra característica importante de ARS que lo diferencia notablemente del resto de algoritmos es que, mientras el resto de algoritmos usan *Deep Learning* con muchas capas ocultas, ARS utiliza una estructura más sencilla por lo que hay menos pesos que ajustar y aprender. Con esta estructura simplista, ARS consigue obtener mayores recompensas en menos tiempo de entrenamiento, pero solo para algunas tareas concretas [33].

Todas estas características hacen de ARS un algoritmo que en tareas concretas llega a ser hasta 15 veces más rápido que sus competidores directos [32]. Este algoritmo debe contemplarse como una opción interesante a tener en cuenta en cualquier tarea en la que queramos aplicar RL, puesto que pueden pasar dos cosas: que la tarea no sea compatible

con el algoritmo y no se obtengan buenos resultados, por lo que otros algoritmos como TRPO, PPO, etc darán resultados mejores; o bien puede darse el caso de que la tarea sea compatible con el algoritmo y se obtengan resultados mucho mejores que para el resto de algoritmos.

En el apartado de experimentación, trataremos de comprobar si las características descritas a lo largo de este apartado son ciertas o en que medida se corresponden para el entorno y tarea que es este proyecto se estudian.

Capítulo 5

Entornos de simulación

En esta sección se describen cada uno de los entornos de experimentación que se utilizan en este proyecto y sus características más importantes.

Uno de los pilares fundamentales para conseguir que el agente de RL aprenda es tener un entorno bien diseñado que proporcione realimentación de calidad al agente sobre cómo de buenas son las acciones que realiza, y de este modo que el aprendizaje se realice de la forma más rápida y correcta posible. En este trabajo se han utilizado cuatro entornos proporcionados por el repositorio S-RL Toolbox. Estos entornos podremos visualizarlos gracias a la herramienta Gym para obtener así una realimentación visual y comprobar los resultados obtenidos.

En todos los entornos tenemos como agente ejecutor de acciones al robot **Kuka LBR iiwa**, situado sobre una mesa a 0.5 metros del suelo. Las diferentes tareas son todas también episódicas, ya que el robot parte de una posición inicial y la tarea termina cuando se alcanza la posición objetivo. EL robot inicia cada episodio siempre con sus articulaciones rotacionales con valores de (0,0,90,0,0,90) grados, lo que corresponde a todas las articulaciones con ángulo cero excepto las tercera y la quinta, que marcan un ángulo de 90 grados.

5.1. Botón fijo

En este entorno el objetivo del robot es accionar un botón de 3 centímetros de radio situado encima de la misma mesa que sujeta la base del robot, a aproximadamente 20 centímetros delante de este. La posición del botón es fija, es decir, es la misma para

todos los episodios.

Los estados que marcan el final del episodio son:

- El robot acciona el botón 5 veces o lo mantiene durante 5 segundos (cada pulso de un segundo cuenta como un accionamiento), lo que se recompensará con +1 por cada accionamiento. La recompensa máxima que puede alcanzar el agente es de +5, lo que marcara el final de la tarea.
- El robot golpea contra la mesa o supera el umbral de distancia máxima respecto al objeto, lo que se recompensará con -1. En el caso de que el robot haya accionado alguna vez el botón previamente, la recompensa total entregada al agente será la suma de las recompensas obtenidas a lo largo del episodio.
- Para evitar que el robot quede estancado infinitamente en un estado intermedio, se establece un número límite de 1000 pasos en cada episodio. De este modo, en el caso de que ninguno de los estados finales se produzca en el periodo marcado, el episodio terminará entregando una recompensa nula al agente.

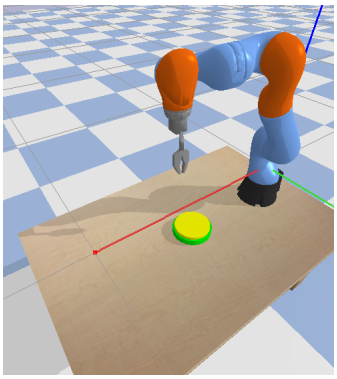

Figura 5.1: Entorno de botón fijo (simulador Gym)

5.2. Botón aleatorio

En este entorno el botón ya no se encuentra fijo. En cada nuevo episodio su posición es escogida aleatoriamente, pero siempre sobre el plano de la mesa y en un área contenida en el espacio de trabajo del robot.

Los estados finales de los episodios y las recompensas asociadas son las mismas que en el entorno de botón fijo.

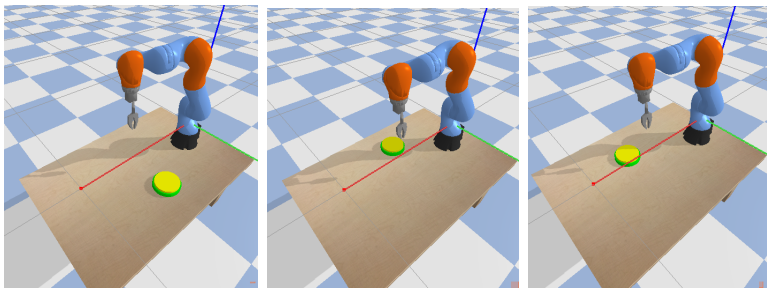

Figura 5.2: Sucesión de imágenes del estado inicial en diferentes episodios (simulador Gym)

5.3. Botón móvil aleatorio

En este entorno se presenta un solo botón que se mueve de derecha a izquierda en línea recta, siendo aleatoria la posición inicial de este y la dirección inicial de movimiento. El rango de movimiento del botón está limitado dentro del espacio de acción del robot, y su movimiento siempre será de un extremo a otro, es decir sin cambios de sentido a mitad del recorrido. El robot debe aprender que el objetivo está cambiando su posición constantemente a lo largo de todo el episodio y deberá adaptar su posición en consecuencia (haciendo un seguimiento o *tracking*).

Los estados que marcan el final del episodio y las recompensas asociadas son las mismas que en el entorno del botón fijo, pero con la diferencia de que en este caso el límite de pasos por episodio también es 1500.

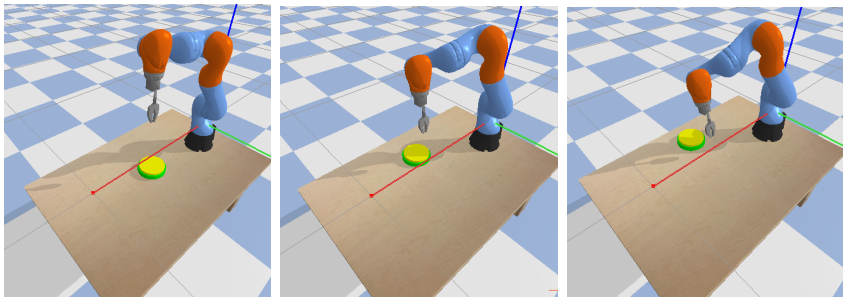

Figura 5.3: Sucesión de imágenes durante un episodio en el que se puede apreciar el movimiento lateral del botón (simulador Gym)

5.4. Dos botones fijos

Ante el robot se presentan dos botones, uno al lado del otro, uno de color amarillo y otro de color verde oscuro. Las posiciones de los botones son fijas y comunes en todos los episodios. El objetivo es que el robot aprenda la secuencia de activación correcta: primero debe pulsar el botón amarillo y luego el botón verde oscuro, obteniendo una recompensa de +1 por cada activación.

Los estados que marcan el final del episodio son:

- El robot acciona ambos botones en la secuencia correcta, obteniendo una recompensa total de +2.
- El robot golpea contra la mesa o supera el umbral de distancia máxima respecto al objeto, lo que se recompensará con -1. Si bien, antes de que esto suceda, el robot podría activar el primer botón, por lo que la recompensa total quedaría en cero.
- Para evitar que el robot quede estancado infinitamente en un estado intermedio, se establece un número límite de 1500 pasos en cada episodio. De este modo, en el caso de que ninguno de los estados finales se produzca en el periodo marcado, el episodio terminará entregando una recompensa nula al agente.

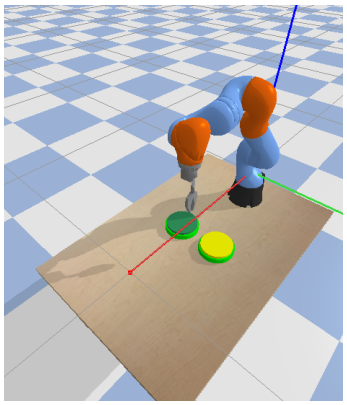

Figura 5.4: Entorno de dos botones (simulador Gym)

Los entornos planteados se han escogido teniendo en cuenta una serie de características que garanticen una buena base sobre la que realizar proyectos de investigación y comparación de resultados. Estas características son:

- Se trata de tareas básicas para un brazo robótico, en las que el objetivo consiste

en alcanzar posiciones concretas del espacio.

- Los entornos son simples. Contamos con pocos elementos relevantes (el robot y el objetivo), lo que facilita la interpretación de información mediante visión.
- Cada uno de los entornos planteados presenta un grado de dificultad mayor que el anterior.

Capítulo 6

Experimentación

Los primeros pasos que llevamos a cabo consistieron en preparar nuestro ordenador con el software necesario para poder llevar a cabo los experimentos.

En primer lugar, descargamos e instalamos Anaconda en nuestra máquina. Seguidamente descargamos el repositorio de S-RL Toolbox [35] y creamos un entorno virtual de Anaconda en el que instalar todas las dependencias necesarias para el correcto funcionamiento de los programas. Basta con abrir el terminal y ejecutar el siguiente comando:

```
conda env create --file environment.yml
```

Donde el fichero `environment.yml` es un documento en el que se encuentran listadas todas las dependencias y paquetes pip, y también el nombre que le queremos dar al entorno (en nuestro caso py35). Para activar el entorno que acabamos de crear debemos ejecutar:

```
        $source activate py35$
```

En principio, con estos pasos podemos empezar a trabajar con el Toolbox, si bien, dado que se trata de una herramienta compleja que requiere de muchas dependencias, es muy posible que en un primer instante encontremos incompatibilidades y errores justo en el momento en que lanzamos cualquier entrenamiento. Esto puede deberse a que alguna de las versiones de paquetes instalados no sean compatibles entre sí o con nuestra máquina. Por ejemplo, debemos considerar si vamos a utilizar la GPU o si vamos a correr los procesos directamente en CPU. En función de esto instalaremos la versión

de tensorflow para CPU o para GPU. Otro paquete que también suele causar problemas es la versión de Stable Baselines. En nuestro caso el archivo `environment.yml` tenía asignada una versión desactualizada de Stable Baselines, y al actualizarlo creó varias incompatibilidades con el resto de dependencias que tuvimos que resolver una a una.

En este proyecto los experimentos han sido ejecutados en una GPU GForce GTX 1080, proporcionada por el Grupo de Reconocimiento de Formas e Inteligencia Artificial perteneciente al departamento de Sistemas y Lenguajes informáticos de la Universidad de Alicante.

6.1. Colocación de la cámara

En primer lugar definimos cual es la posición de la cámara, dado que nuestro sistema hace uso de la visión artificial 2D para obtener información sobre cuál es la posición del objetivo y del extremo de la herramienta del robot dentro del entorno.

Puesto que los experimentos se han realizado con una sola cámara dada la alta complejidad que supondría añadir más de una, es indispensable escoger una perspectiva que nos proporcione la mayor información posible. Este hecho determinará en gran medida la velocidad y calidad del aprendizaje.

Por ejemplo, si colocamos la cámara en una posición elevada y con poco ángulo, casi encima del robot, como la que se muestra en la imagen 6.1, apenas tendremos información de altura y al robot le será mucho más complicado aprender cómo llegar al objetivo. En la imagen 6.2 podemos apreciar que, visto desde arriba el robot parece que esté prácticamente en la misma posición que en la imagen 6.1, pero si cambiamos el punto de vista a una visión lateral se ve claramente que hay una diferencia muy importante que no estamos percibiendo.

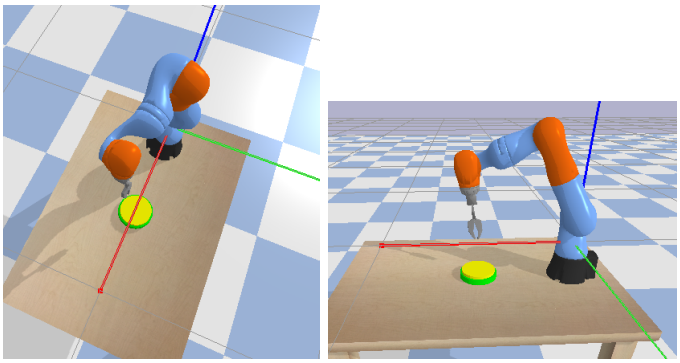

Figura 6.1: Posición 1, vista superior y lateral

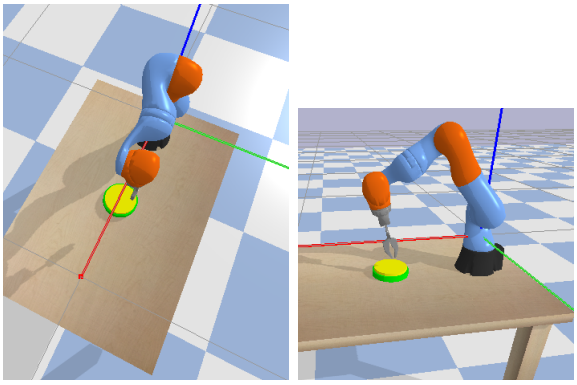

Figura 6.2: Posición 2, vista superior y lateral

Del mismo modo, para el caso contrario, si elegimos un punto de vista totalmente lateral al robot, tenemos mejor información de altura pero ninguna información de profundidad. Esto también se puede apreciar en las imágenes anteriores, donde desde la vista lateral vemos si el robot está más arriba o más abajo, pero no sabemos si el extremo del robot está alineado con el botón.

La imagen 6.3 muestra el punto de vista elegido para realizar los experimentos, simulando la posición de la cámara. Este punto de vista ha sido seleccionado teniendo en cuenta los factores anteriores y tratando de conseguir la mayor información visual posible. Si bien, debemos ser conscientes que seguiremos teniendo problemas de falta de información que solo podríamos solucionar con más cámaras. (aunque más cámaras no garantizan mayor información si estas no están colocadas debidamente).

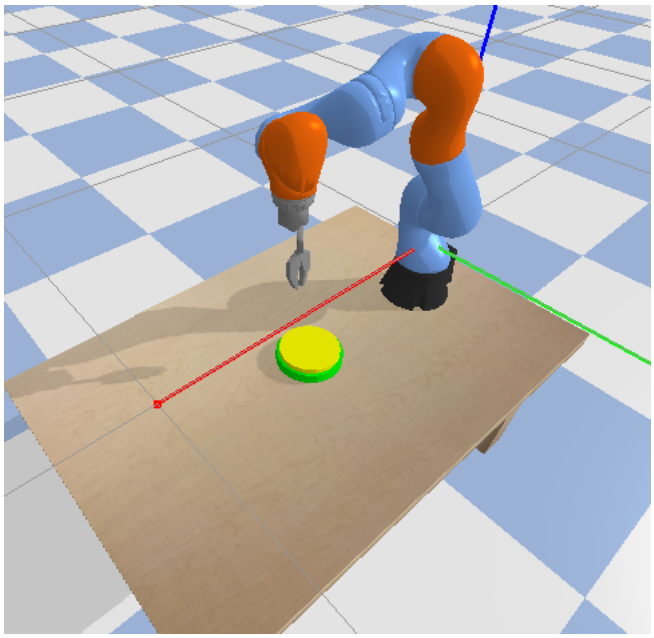

Figura 6.3: Posición de la cámara durante los experimentos

La información que podamos obtener mediante la visión va muy ligada al espacio de acciones y el espacio de estados que definamos en nuestro sistema. Ya hemos comprobado que si nos basamos únicamente en la imagen de la cámara para definir en que estado se encuentran el agente y el objetivo, en muchas ocasiones ésta nos dará la misma información para dos posiciones del robot diferentes, debido a que nos falta la información de profundidad. Eso hace que sea mucho más difícil de entrenar, dado que ¿Cómo puede aprender el agente que para el mismo estado en unas ocasiones debe aplicar una acción y en otras ocasiones debe aplicar otra acción totalmente distinta?. Por ejemplo, puede que con la misma imagen en un caso el robot esté encima del botón y en el otro esté a la misma altura pero más adelantado.

En muchas ocasiones, la representación de la información proporcionada por los sensores tiene una dimensión mayor que la información que es realmente útil para definir el espacio de estados de una tarea. La ventaja de RL es que nos permite aprender una tarea directamente de la información en crudo, teniendo en cuenta que cuanta más información haya más tiempo de entrenamiento se necesitará. Esto es justo lo que se hace en este proyecto, el agente aprende de un modo *end-to-end*, es decir, aprende a identificar cada estado directamente a partir de la información de la cámara. Diremos que cada *frame* de la cámara es la observación en cada instante, mientras que el espacio de estados estará

definido por la información que el propio agente aprende a distinguir como útil para realizar la tarea.

En cuanto al espacio de acciones, S-RL Toolbox nos permite elegir si queremos realizar el aprendizaje del control del robot de dos maneras:

- Siendo el espacio de acciones cada una de las articulaciones del robot. El agente aprende a identificar cuáles de sus 6 articulaciones activa y en qué sentido de giro para poder llegar al objetivo.
- Siendo el espacio de acciones el movimiento de su TCP (*Tool Center Point*) en las direcciones XYZ de un sistema de coordenadas cartesiano. El simulador se encargará de aplicar la cinemática inversa para traducir los movimientos cartesianos en movimientos articulares. El agente aprende a identificar la dirección que debe tomar su TCP para llegar al objetivo.

El principal inconveniente de la primera opción es que diferentes movimientos articulares pueden llevar al TCP al mismo punto, lo que quiere decir que el agente se verá constantemente en la situación de tener varias acciones válidas para un mismo estado. Lidiar con esta situación es una cosa más que debería aprender. Además, el espacio de acciones es más amplio, por lo que se requerirá más tiempo y memoria hasta aprender la tarea. Es por ello que para los experimentos de este proyecto hemos optado por la segunda opción, ya que en este caso es el propio simulador Gym quien se encarga de solucionar el problema de la cinemática inversa.

Una vez elegido el punto de vista de la cámara y definido el espacio de estados y el espacio de acciones con el que trabajará nuestro agente, ya estamos más cerca de poder comenzar con el entrenamiento. Sin embargo, todavía queda concretar los hiperparámetros que utilizaremos en cada algoritmo. Este punto se abarca en el siguiente apartado.

6.2. Búsqueda de hiperparámetros

Uno de los aspectos más complejos que suele ir unido al uso del aprendizaje automático es el ajuste de los hiperparámetros que estos métodos utilizan, ya que para cada entorno y cada tarea el algoritmo puede variar considerablemente sus resultados si usamos unos u otros valores. Para saber usar correctamente estas variables normalmente es necesario tener un alto conocimiento del algoritmo y de como este se comporta en diferentes situaciones.

El método más común para ajustar los hiperparámetros es realizar una cadena de experimentos hasta encontrar aquellos valores que proporcionen los mejores resultados. Este procedimiento es largo y tedioso, y aun más en nuestro caso, dado que en este proyecto se emplean 6 algoritmos para 4 tareas diferentes. Por suerte, S-RL Toolbox nos proporciona una herramienta que nos facilita encontrar buenos valores para los hiperparámetros más importantes de cada uno de los algoritmos estudiados. Esto nos facilita enormemente el trabajo ya que ajustar tantos hiperparámetros podría ser todo un nuevo reto.

S-RL Toolboox realiza un procedimiento muy sencillo: inicialmente, se varían de forma ordenada los parámetros del algoritmo en cuestión, ejecutándolo en cada caso ante entrenamientos cortos (30000 pasos para los experimentos de este proyecto) para quedarnos con los valores que mayor recompensa hayan obtenido; seguidamente, se realiza un procedimiento similar, esta vez partiendo directamente de los mejores valores obtenidos en la primera fase para intentar optimizarlos. Para ello, se realizaran variaciones más pequeñas de los hiperparámetros pero en un conjunto de valores mucho más pequeño. Además, se va aumentando el tiempo de entrenamiento hasta llegar a los 1000000 pasos.

Puesto que este proyecto no tiene el objetivo de entrar más en detalle en este aspecto, agradecemos a RL-Toolbox por proporcionarnos esta herramienta. A continuación, se muestran los valores de los hiperparámetros obtenidos con dicha herramienta para cada algoritmo:

Hiperparámetro	**Valor**
batch_size	10
exploration_final_eps	0.4375872112626925
exploration_fraction	0.8121687287754932
gamma	0.925596638292661
learning_rate	0.04799771723750573
learning_starts	6754
target_network_update_freq	2907
train_freq	2

Tabla 6.1: Hiperparámetros obtenidos para DQN.

Hiperparámetro	**Valor**
alpha	0.9627983191463305
ent_coef	0.07103605819788694
epsilon	0.0056804456109393235
gamma	0.5435646498507704
learning_rate	0.002021839744032572
max_grad_norm	2.691585106789232
n_steps	59
vf_coef	0.7917250380826646

Tabla 6.2: Hiperparámetros obtenidos para A2C.

Hiperparámetro	**Valor**
cg_damping	0.45615033221654855
cg_iters	3
entcoeff	0.10590760718779213
gamma	0.7586156243223572
lam	0.1433532874090464
max_kl	0.4736004193466574
timesteps_per_batch	671
vf_iters	8
vf_stepsize	0.0036071889969380372

Tabla 6.3: Hiperparámetros obtenidos para TRPO.

Hiperparámetro	**Valor**
cliprange	0.6458941130666561
ent_coef	0.5448831829968969
gamma	0.5488135039273248
lam	0.6027633760716439
learning_rate	0.007027629280620722
max_grad_norm	0.7151893663724195
n_steps	1528
noptepachs	5
vf_coef	0.4236547993389047

Tabla 6.4: Hiperparámetros obtenidos para PPO2.

Hiperparámetro	Valor
alpha	0.7894611937694473
correction_term	7.132532595434541
delta	1.952595828333363
ent_coef	0.9564057227959488
gamma	0.7680887473517259
learning_rate	0.009226017040693325
max_grad_norm	2.826504193239529
n_steps	5
q_coef	0.39267567654709434
replay_ratio	4
rprop_alpha	0.5644302827331601
rprop_epsilon	0.002775960977317661

Tabla 6.5: Hiperparámetros obtenidos para ACER.

Hiperparámetro	Valor
exploration_noise	0.0033304626546696195
max_step_amplitude	26.60972262205136
step_size	0.08490383084285108
top_population	3

Tabla 6.6: Hiperparámetros obtenidos para ARS.

6.3. Entrenamiento

Para realizar los entrenamientos hacemos uso del programa `train.py` que S-RL Toolbox [13] nos proporciona, y que a su vez está desarrollado gracias a RL-Baselines [17]. Este programa es genérico para todos los algoritmos de RL que utilizaremos en este proyecto. Es el encargado de recibir los parámetros de ejecución que se utilizarán en el entrenamiento (número de pasos, algoritmo a utilizar, tarea que el agente debe realizar, valores de los hiperparámetros, etc) y filtrar que no haya información errónea o parámetros incompatibles. Seguidamente se realiza una llamada al algoritmo. Este programa quedará en ejecución paralela, a la espera de ir recibiendo las recompensas que el agente consiga durante el entrenamiento.

El código de `train.py` calcula la media de las cien últimas recompensas recibidas hasta el momento tras cada episodio. Esta media la muestra por pantalla denominándola *mean reward*. De este modo podemos tener una idea de cómo se desarrolla el entrenamiento.

Por ejemplo, para el entorno del botón fijo, la ganancia máxima que podemos obtener si el robot completa la tarea con éxito es 5, por lo que si `train.py` nos indica que *mean reward* tiene un valor muy próximo a 2.5 querrá decir que el robot ha completado la tarea con éxito la mitad de las cien últimas veces. Cuanto más se acerque el valor de *mean reward* a 5, querrá decir que mejor se comporta el agente.

Otra variable importante es *best mean reward*, que corresponde con el mejor valor de *mean reward* hallado hasta el momento. Cada vez que *best mean reward* se actualice, `train.py` guardará en un fichero `.pkl` el modelo del agente con el que se ha alcanzado dicho resultado. Debemos tener en cuenta que si el modelo se actualizase cada episodio, a pesar de si *best mean reward* mejora o no, nos veríamos expuestos a perder buenos modelos.

El modelo del agente es como su "cerebro". Dependiendo de qué arquitectura siga el algoritmo, su modelo se corresponderá con los pesos de los parámetros de la política, los pesos de la red neuronal, o de la función de valor. En todos los casos es lo mismo, dentro del modelo tenemos los parámetros que definen el comportamiento del agente.

Algo a lo que también debemos de prestar atención es que no siempre un *mean reward* muy alto significa que el agente ha aprendido mucho, principalmente en el inicio del entrenamiento, cuando se han ejecutado solo unos pocos episodios. Imaginemos, para el entorno del botón fijo, que al lanzar un entrenamiento de mil episodios, para los tres primeros episodios: en los dos primeros se cumple el tiempo máximo y el robot ni alcanza el objetivo ni choca contra la mesa (la recompensa es cero), y en la tercero, por casualidad, el robot alcanza el objetivo; en este caso el valor medio de las recompensas sería 1.6667, que hasta el momento sería el mejor *mean reward* obtenido, por lo que se asignaría dicho valor a *best mean reward* y guardaríamos el modelo como válido. Si bien, puesto que hemos alcanzado el objetivo por casualidad, puede que el modelo guardado no sea realmente adecuado y no nos lleve de nuevo a alcanzar el objetivo. Después de estos tres episodios, en el entrenamiento el robot no alcanza el objetivo durante los ochocientos noventa y siete siguientes episodios. A partir de ese momento el agente empieza a aprender realmente su tarea y acierta el 40 % de los últimos cien intentos y el resto de intentos choca contra la mesa, quedándose cerca del objetivo. Parece obvio que el modelo obtenido en la última parte del entrenamiento es mejor que el obtenido en el tercer episodio, sin embargo, al final del entrenamiento el *mean reward* es, como máximo 1.4, mientras que en el tercer episodio teníamos 1.67. Como el *mean reward* obtenido al final del entrenamiento no supera el *best mean reward* el sistema no considera el segundo modelo mejor que el primero, por lo que no se guarda y terminamos el entrenamiento

con un agente que realmente no ha aprendido nada.

Ante el planteamiento de este problema podría pensarse que una posible solución sería quedarnos siempre con el modelo obtenido al final del entrenamiento. Sin embargo esto tampoco sería la solución, dado que podría ocurrir que en los primeros instantes del entrenamiento el modelo alcance una solución local con unos pesos que son buenos que proporcionan resultados razonablemente buenos, pero a al final del entrenamiento, tratando de llegar a una solución mejor, el modelo se encuentre con un valle de gradiante y se acabe quedando con un resultado peor.

La solución que se ha puesto a este problema ha sido poner un umbral mínimo de episodios que deben ejecutarse hasta asignar por primera vez un valor a *best mean reward* al que mantendremos con un valor constante -10000. Por tanto, una vez superemos el umbral, en el momento que *mean reward* supere un valor de -10000, se asignará dicho valor a *best mean reward* y se guardará el modelo actual en un fichero `.pkl`.

Tras cada entrenamiento, los resultados y parámetros empleados se guardan en los siguientes ficheros:

- Un fichero `.pkl` que almacena el modelo del agente.
- Un fichero `0.monitor.csv` en el que se visualiza para cada episodio la recompensa obtenida, el número de pasos empleados, y el tiempo que ha transcurrido desde el inicio del entrenamiento hasta el final del episodio actual.
- `args.json`, en el que se guardan los argumentos seleccionados para llevar a cabo el entrenamiento: algoritmo, número de pasos, hyperparámetros, tipo de aprendizaje (*end-to-end*), umbral de episodios a ejecutar antes de guardar el modelo por primera vez, etc
- `env-globals.json`, donde se guardan los valores de las variables globales que definen el entorno en el que se ha ejecutado el entrenamiento: posición del botón, velocidad de desplazamiento y sus posiciones máxima y mínima, posición de la cámara, número de veces que hay que pulsar el botón para que el entrenamiento termine, etc.
- `rl-locals.json`, en el que se guarda información del estado de nuestra máquina en el momento en que se lanzó el entrenamiento y otros parámetros que son menos relevantes de entender y que se salen de los objetivos de este proyecto.

6.4. Resultados y evaluación

A continuación se muestran una serie de gráficas para representar los resultados obtenidos en los experimentos ejecutados. En cada gráfica se muestra la evolución del *mean reward* a lo largo del entrenamiento. Cada entrenamiento se ha realizado durante 10^6 pasos, y puesto que no todos los episodios requieren del mismo número de pasos, el número total de episodios realizados en cada entrenamiento es distinto.

Tal y como ya se ha mencionado en la introducción, en este apartado se pretende mostrar y comprender el comportamiento de diferentes algoritmos de RL y Deep RL ante problemas de complejidad diversa. Unido a los conceptos plasmados en el apartado de Metodología, se esperan obtener conclusiones de forma razonada y justificada que ayuden a conocer mejor el resultado de aplicar RL en problemas simples de robótica industrial.

Otro aspecto en el que también indagaremos es en si el grado de dificultad que intuitivamente asignamos a cada tarea se corresponde a la dificultad que encuentran los algoritmos. Es decir, si ordenamos las tareas intuitivamente de menor a mayor dificultad, el orden sería: botón fijo, botón aleatorio, botón móvil aleatorio y dos botones fijos. En los experimentos comprobaremos si este orden se cumple o si hay tareas que el agente aprende con mayor rapidez que otras sin ser esto lo esperado.

Para poder obtener resultados que realmente representen el comportamiento de cada algoritmo ante cada entorno, cada entrenamiento se ha repetido tres veces, y la media de esas tres ejecuciones se representa en las siguientes gráficas.

Entorno de botón fijo

En la figura 6.4 podemos observar como en los 10.000 primeros pasos todos los algoritmos alcanzan rápidamente un *mean reward* superior a 3. Que los agentes aprendan tan rápido puede ser debido a que la tarea es muy sencilla y repetitiva. Aprender esta tarea realmente no aporta genericidad, el agente se especializa en una situación concreta ya que el botón está siempre en el mismo punto.

Por un lado tenemos a los algoritmos A2C, ACER, ARS y PPO que proporcionan resultados muy buenos, llegando a alcanzar todos ellos un *mean reward* máximo 5 en varios momentos a lo largo del entrenamiento. Además, se mantienen bastante constantes a lo largo del proceso, a excepción de ACER, que oscila en un *mean reward* de entre 4

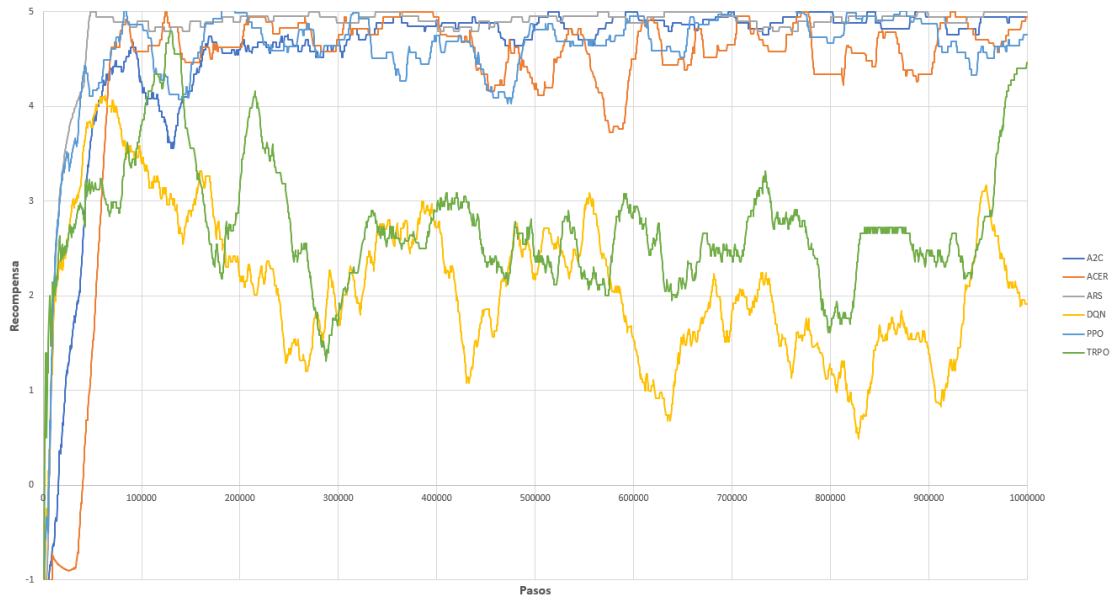

Figura 6.4: Resultados de los entrenamientos con el entorno de botón fijo.

y 5, incrementando esta inestabilidad en la segunda mitad del entrenamiento.

Por otro lado, tenemos los resultados de DQN y TRPO. Estos, a pesar de tener un rápido aprendizaje en la primera parte del entrenamiento, rápidamente empiezan a desaprender y permanecen todo el entrenamiento con resultados inestables. En este caso, el peor resultado lo proporciona DQN, el cual a pesar de haber llegado a un *mean reward* superior a 4 en los primeros 60.000 pasos, decae hasta 0,46 en 82.500 pasos.

En este entorno destaca especialmente ARS, que solo necesita una media de 54.301 pasos para alcanzar un *mean reward* de 5. También destaca su gran estabilidad a lo largo del entrenamiento, siendo el que más tiempo permanece con un *mean reward* de 5.

Entorno de botón aleatorio

En el entorno de botón aleatorio, cuyos resultados pueden verse en la figura 6.5, se confirma que en el caso anterior se obtenían resultados muy buenos porque los agentes estaban sobre-aprendiendo una tarea monótona por definición. En este caso, ninguno de los algoritmos alcanza un *mean reward* superior a la unidad al final del entrenamiento. ARS es de nuevo el que mejor comportamiento muestra, manteniendo sus resultados por encima del resto durante casi todo el entrenamiento y siendo el único que supera un *mean reward* de 1 en varias ocasiones. Sin embargo, al contrario que en el primer

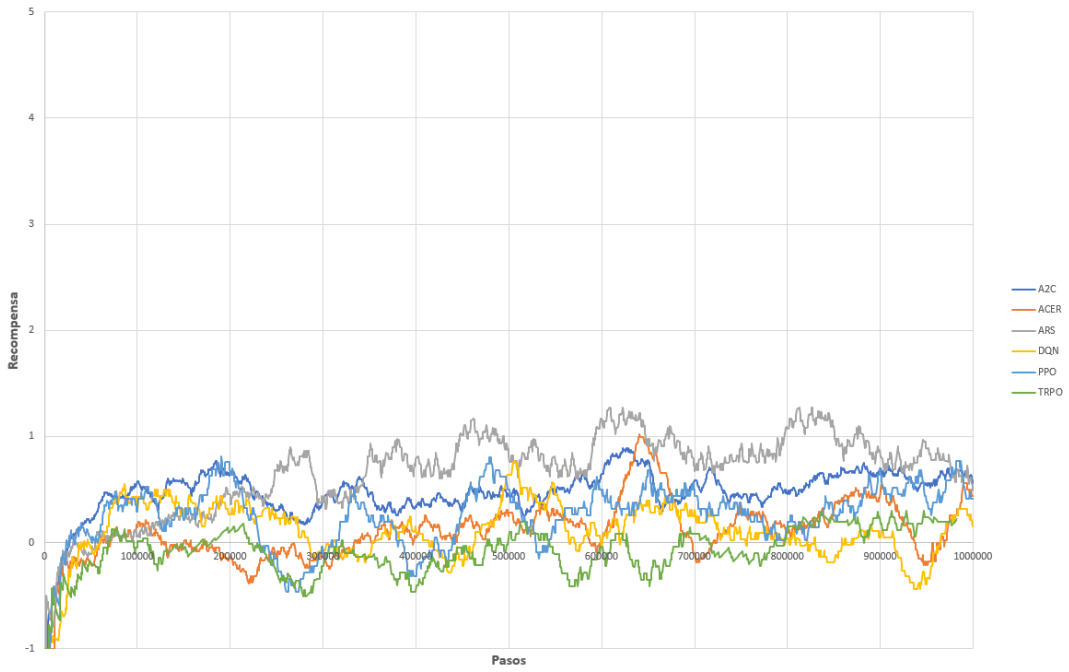

Figura 6.5: Resultados de los entrenamientos con el entorno de botón aleatorio.

entorno, es uno de los que tiene una subida inicial más suave.

Por otro lado, PPO vuelve a mostrar resultados buenos y es el que más estable se mantiene a lo largo del entrenamiento.

Con peores resultados al inicio del entrenamiento destacan TRPO y ACER, a los que les cuesta bastante mantenerse con un *mean reward* positivo. Si bien, con ACER se alcanza un pico que llega hasta la unidad en 63.905 pasos, cosa que solo ha conseguido ARS en este entorno.

En términos generales, los resultados en este entorno son peores de lo esperado (con recompensas medias entorno al 0,6 al final del entrenamiento para la mayoría de los algoritmos) teniendo en cuenta que inicialmente se planteaba como el segundo entrono menos complejo.

Entorno de botón móvil

Lo primero que sorprende del entorno de botón móvil cuyos resultados pueden verse en la figura 6.6 es que para todos los agentes se han obtenido resultados mejores que en el entorno del botón con posición aleatoria, que suponíamos más sencillo.

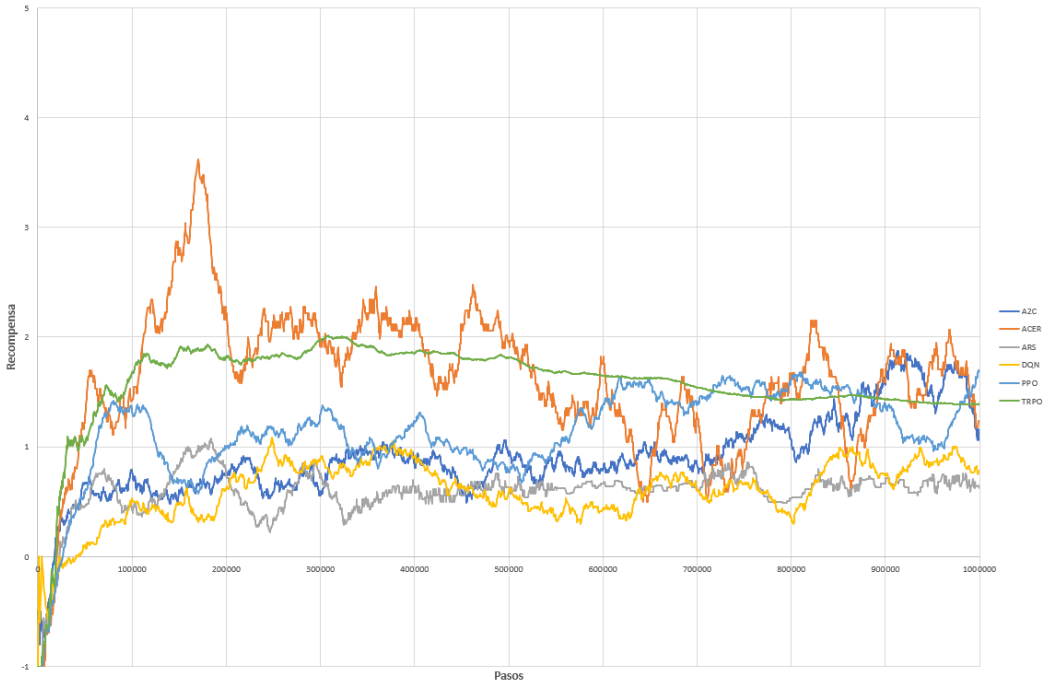

Figura 6.6: Resultados de los entrenamientos con el entorno de botón móvil.

En lo positivo, destacan principalmente los resultados de ACER y TRPO. El primero por tener una subida inicial de aprendizaje que alcanza valores medios de recompensa de hasta 3,55, pero que se mantiene muy inestable a lo largo del entrenamiento y por ello termina con recompensas medias similares a algoritmos con aprendizajes más lentos. En cuanto a TRPO, este sorprende con un comportamiento muy estable y bueno a lo largo de todo el proceso.

De nuevo PPO se mantiene en la media entre los mejores y peores algoritmos, con un aprendizaje lento pero constante, al igual que A2C, que llega hasta un *mean reward* de 1,88 poco antes del final del entrenamiento.

Al igual que en los entornos anteriores DQN es uno de los que peores resultados obtiene. Pero quien en este caso tiene un peor comportamiento es ARS, lo cual sorprende bastante teniendo en cuenta que ha destacado por sus buenos resultados en los casos anteriores.

Que los resultados en este entorno sean mejores que en el caso anterior puede deberse al propio movimiento del botón. En el caso del botón colocado en posiciones aleatorias el agente puede no tener claro donde está colocado debido a sus limitaciones visuales (utilizamos visión 2D). Si bien, si el botón se mueve sobre la mesa, la diferencia de

su posición respecto de la mesa entre fotogramas permite una mejor percepción de la posición del botón. Esto es muy sencillo de entender con un simple ejemplo práctico: si ponemos una moneda frente a nosotros encima de la mesa en una posición aleatoria, nos tapamos un ojo y tratamos de tocarla con el dedo partiendo este de una posición elevada, nos será más difícil acertar si la moneda está fija que si se está moviendo.

Entorno de dos botones

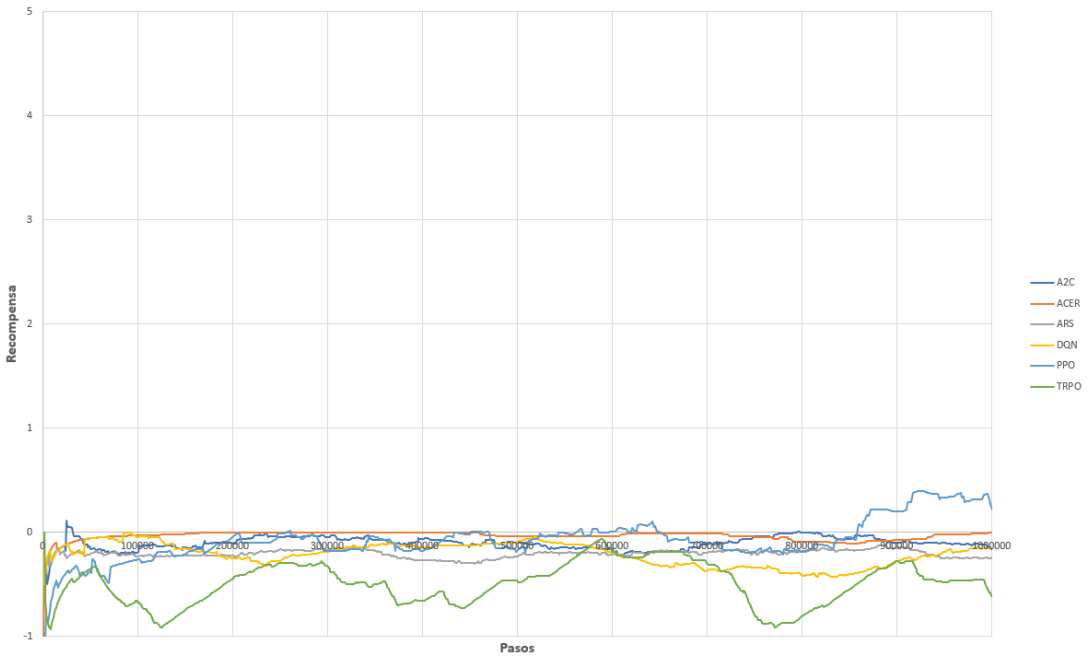

Figura 6.7: Resultados de los entrenamientos con el entorno de dos botones.

En el entorno de dos botones (ver figura 6.7) vemos que los algoritmos no son capaces de aprender la tarea. Las razones de esto pueden ser:

- Que el tiempo de entrenamiento sea demasiado corto para la alta complejidad de esta tarea.
- Que las recompensas no estén bien definidas. Hay que tener en cuenta que la recompensa por alcanzar el primer botón es +1, y si el TCP del robot choca contra la mesa o se sale de rango obtenemos -1. Por tanto, si el robot toca el botón y luego choca contra la mesa la recompensa total será nula, por lo que el agente no aprenderá nada.

A2C es el único que alcanza recompensas positivas, llegando a un *mean reward* máximo de 0,41 en el último tramo del entrenamiento.

Al igual que en los dos primeros entornos, vuelven a destacar TRPO y DQN por tener los peores resultados.

Si hacemos una evaluación general de los resultados y tratamos de relacionarlos con la teoría vista en el apartado de Metodología, podemos destacar los siguientes puntos:

- DQN ha sido el que peores resultados ha proporcionado en dos de los entornos y el segundo peor en los otros dos. Es posible que esto sea debido a que se trata de unos de los algoritmos más sencillos que aquí se plantean. Por lo que estos experimentos han servido para descartarlo en aplicaciones de índole robótica.
- Los experimentos han demostrado las ventajas de PPO. Es un algoritmo que aporta genericidad, donde sus resultados han sido razonables para las distintas tareas, sin destacar por ser el mejor ni el peor en ninguna de ellas, compitiendo e incluso superando a alternativas de implementación más complejas como es TRPO.
- A2C nos deja conclusiones similares a PPO, un algoritmo que se adapta a diferentes tareas. Gracias a esto, entendemos que ambos representen el estado del arte de las dos arquitecturas más importantes en RL: PG y *Actor-Critic*.
- TRPO, pese a destacar por ser un algoritmo de alta complejidad, solo ha proporcionado resultados razonables en el entorno del botón móvil. Con esto, podemos afirmar la advertencia de OpenAI que decía que este algoritmo no es compatible para resolución de tareas con entrada visual significativa.
- ACER ha destacado por tener resultados bastante inestables, si bien, en la mayoría de casos obtiene picos de aprendizaje que superan a la mayoría de algoritmos. Estos picos son interesantes ya que debemos recordar que el modelo del agente se guardará en el momento en que se llega al mejor *mean reward*. También se confirma que ACER es una mejora a TRPO dado que sus resultados son mejores en todos los entornos.
- ARS se ha ajustado a lo predicho en la metodología, donde advertíamos que era un algoritmo que proporcionaba muy buenos resultados en tareas concretas y muy malos en otras. Sus resultados han sido los mejores en los entornos de botón fijo y botón aleatorio, pero ha sido el peor método en el entorno de botón móvil. Si bien, en los mejores casos su diferencia con el resto no ha sido tan significativa como se esperaba.

Otra conclusión importante es que los entornos no han seguido el orden de dificultad esperado, ya que el entorno en el que el botón toma posiciones aleatorias ha resultado más difícil de aprender que el entorno en el que el botón se mueve.

Con los experimentos realizados no podemos decir que RL sea una técnica sencilla para aplicar a un brazo robótico industrial de forma efectiva. Es posible que se necesiten entrenamientos mucho más largos y una mejor definición de las recompensas.

Capítulo 7

Conclusiones

Este proyecto ha sido una pieza clave con la que interiorizar muchos de los conceptos de inteligencia artificial vistos a lo largo del grado de Ingeniería Robótica. Durante su elaboración se han alcanzado los objetivos propuestos. El alumno ha sido capaz de comprender los pilares sobre los que se asienta RL e indagar en varios de los algoritmos que conforman el estado del arte en este campo.

S-RL Toolbox ha resultado en una herramienta clave con la que realizar experimentos y poner en práctica los conocimientos. Gracias a la buena estructuración de esta librería y a su implementación en python, el alumno ha sido capaz de comprender el código y emplear los programas para realizar los experimentos planteados.

El simulador Gym ha resultado también clave para realizar la evaluación. De no ser por este simulador, los experimentos se hubieran vuelto inviables dada su complejidad y el tiempo de cálculo. Además, poner a un robot real a realizar movimientos casi aleatorios al inicio de un aprendizaje acarrea una alta peligrosidad y podría tener graves consecuencias. En este proyecto se ha comprobado que este tipo de software es esencial en experimentos de este calibre.

También hemos comprobado como RL nos permite alimentar la entrada del sistema directamente con datos en crudo. Al agente no se le enseña qué debe aprender y cómo debe aprenderlo, él mismo llega a la solución con el simple principio de alcanzar la mayor recompensa posible. La desventaja de este tipo de aprendizaje *end-to-end* es que ha resultado costoso computacionalmente y eso se ha traducido en largas esperas para obtener los resultados de los experimentos, siendo inviable de ejecutar en el PC del alumno.

Tal y como está definida la asignación de recompensa para las tareas planteadas en este proyecto, tenemos la desventaja de que hay que esperar hasta el final de cada episodio para calcular la recompensa. Un error común es deducir que si se obtiene una alta recompensa es porque todas las acciones que se tomaron fueron correctas, incluso si una de ellas fue realmente mala. Como consecuencia, para obtener una política óptima, es necesario realizar un gran número de intentos, lo que ralentiza el aprendizaje. Quizás podría reducirse el tiempo de aprendizaje y mejorar el comportamiento del robot haciendo que se calcule la recompensa en cada *step*, haciendo que ésta sea proporcional a la distancia que hay del TCP al objetivo; de este modo se puede saber en mejor medida cómo de buena es la acción que se toma en cada momento.

El uso de S-RL Toolbox nos ha ayudado a comprender mejor cuál es el procedimiento normal a seguir para aplicar métodos de inteligencia artificial del estado del arte en un robot industrial. Este proyecto deja abiertas muchas puertas a trabajos futuros y ha servido para que el alumno aunara todos los conocimientos vistos a lo largo del grado.

Detrás de lo que en esta memoria ha quedado plasmado, queda mucho tiempo invertido para comprender y depurar la multitud de compleja parametrización que caracteriza a los algoritmos de RL y Deep RL, y que han requerido de un esfuerzo extra en su ajuste para obtener resultados razonables.

7.1. Trabajo futuro

Este proyecto ha sido todo un reto y una gran experiencia para el alumno. Si bien esto no queda aquí, gracias a este trabajo se han abierto muchas puertas de futuras propuestas y ramas de investigación interesantes.

Como no podía ser de otro modo, un paso futuro sería aplicar los modelos a un robot real. Por ejemplo podría ponerse en práctica con los robots industriales ABB IRB 120 con los que cuenta la Universidad de Alicante en el laboratorio de robótica. Gracias a que S-RL Toolbox está implementada en python y al uso de Tensorflow, podríamos utilizar como computadora una tecnología tan sencilla como es la RaspBerry.

En este proyecto se ha puesto el foco de atención sobre la tarea de aprender la trayectoria hacia el objetivo, pero no hay una parte de manipulación, que es realmente el fin que tienen en la industria la mayoría de robots antropomórficos. Es por ello que se deja la puerta abierta a aplicar un segundo módulo también basado en inteligencia artificial, que aprenda a coger el objeto en función de la forma de este y la herramienta que posea

el robot. Este módulo se encargaría de identificar cuáles son los puntos de agarre óptimos del objeto y la apertura y presión que debe aplicar con la pinza.

En cuanto a mejorar los resultados obtenidos, contamos con varias ideas:

- La primera sería utilizar más cámaras para proporcionar más información al agente sobre el entorno. Con una cámara más ya conseguiríamos lidiar con el problema de la falta de información de profundidad y evitaríamos el problema de tener incongruencias como que para la misma imagen de entrada la acción correcta sea diferente. Sin embargo, debe tenerse en cuenta el hecho que al proporcionar mayor información también tenemos mayor coste computacional, pero seguramente mejoraríamos en calidad de aprendizaje.
- Por otro lado, que el agente aprenda de un modo *end-to-end* se convierte en una desventaja cuando queremos llevar este proyecto más allá de la simulación. Cuando se entrena directamente de la información en crudo se tarda mucho más en aprender. Una opción sería lo que plantea S-RL Toolbox, utilizar un sistema de representación aparte. En particular, una red neuronal que aprenda a obtener las mejores características (filtrar la información útil) y pasarle solo eso al agente, de modo que la dimensión de datos con la que aprende es mucho menor y esto aceleraría el aprendizaje, aumentando su efectividad y disminuyendo el coste computacional.

Índice de figuras

Índice de tablas

Bibliografía

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[2] Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence. `https://www.sciencedirect.com/science/article/pii/S0007681318301393`. Accessed: 2019-5-11.

[3] Inteligencia artificial. `https://es.wikipedia.org/wiki/Inteligencia_artificial`. Accessed: 2019-10-24.

[4] Técnicas de inteligencia artificial. `https://rua.ua.es/dspace/bitstream/10045/17323/9/intro_aprendizaje.pdf`. Accessed: 2019-5-11.

[5] S.J. Russell, P. Norvig, and J.M.C. Rodríguez. *Inteligencia artificial: un enfoque moderno*. Colección de Inteligencia Artificial de Prentice Hall. Pearson Educación, 2004.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007.

[7] A Beginner's Guide to Deep Reinforcement Learning. `https://skymind.ai/wiki/deep-reinforcement-learning#define`. Accessed: 2019-5-15.

[8] An introduction to Reinforcement Learning. `https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/`. Accessed: 2019-5-20.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[10] An introduction to Policy Gradients with Cartpole and Doom. `https:`

//www.freecodecamp.org/forum/t/an-introduction-to-policy-gradients-with-cartpole-and-doom/190237. Accessed: 2019-9-18.

[11] An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog! https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/?source=post_page----------. Accessed: 2019-6-7.

[12] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning, 2018. cite arxiv:1811.12560.

[13] Antonin Raffin, Ashley Hill, René Traoré, Timothée Lesort, Natalia Díaz-Rodríguez, and David Filliat. S-rl toolbox: Environments, datasets and evaluation metrics for state representation learning. *arXiv preprint arXiv:1809.09369*, 2018.

[14] Anaconda (distribución de Python). https://en.wikipedia.org/wiki/Anaconda_(Python_distribution). Accessed: 2019-6-10.

[15] Python. https://es.wikipedia.org/wiki/Python. Accessed: 2019-6-27.

[16] Introducción a TensorFlow (Parte 1). https://medium.com/ai-learners/introduccion-a-tensorflow-parte-1-840c01881658. Accessed: 2019-6-27.

[17] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[18] Getting Started with Gym. https://gym.openai.com/docs/. Accessed: 2019-6-10.

[19] Diving deeper into Reinforcement Learning with Q-Learning. https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/. Accessed: 2019-6-22.

[20] Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG). https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287. Accessed: 2019-6-22.

[21] RL — Trust Region Policy Optimization (TRPO) Explained). https://medium.com/@jonathan_hui/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9. Accessed: 2019-9-22.

[22] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. 2000.

[23] The idea behind Actor-Critics and how A2C and A3C improve them. https://sergioskar.github.io/Actor_critics/. Accessed: 2019-10-11.

[24] Proximal Policy Optimization (PPO) with Sonic the Hedgehog 2 and 3. https://towardsdatascience.com/proximal-policy-optimization-ppo-with-sonic-the-hedgehog-2-and-3-c9c21dbed5e. Accessed: 2019-8-15.

[25] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[26] Trust Region Policy Optimization. https://spinningup.openai.com/en/latest/algorithms/trpo.html. Accessed: 2019-9-14.

[27] Proximal Policy Optimization. https://openai.com/blog/openai-baselines-ppo/. Accessed: 2019-9-22.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[29] Introduction to Various Reinforcement Learning Algorithms. Part II (TRPO, PPO). https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-part-ii-trpo-ppo-87f2c5919bb9. Accessed: 2019-9-11.

[30] RL — Proximal Policy Optimization (PPO) Explained. https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12. Accessed: 2019-8-15.

[31] Comparison and Selection of RL Algorithms in Continuous Action Spaces. https://www.reddit.com/r/reinforcementlearning/comments/8w7mn2/comparison_selection_of_rl_algorithms_in/. Accessed: 2019-10-11.

[32] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *CoRR*, abs/1803.07055, 2018.

[33] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016.

[34] Augmented Random Search —One of the Best RL Algs + What I Built. https://hackernoon.com/augmented-random-search-one-of-the-best-rl-algs-what-i-built-e0e3e765808a. Accessed: 2019-8-28.

[35] S-RL Toolbox - github. https://github.com/araffin/robotics-rl-srl. Accessed: 2019-5-11.